NI MATRIXx™

AutoCode™ Reference

April 2007
370768C-01

NATIONAL INSTRUMENTS™

Worldwide Technical Support and Product Information

ni.com

National Instruments Corporate Headquarters

11500 North Mopac Expressway Austin, Texas 78759-3504 USA Tel: 512 683 0100

Worldwide Offices

Australia 1800 300 800, Austria 43 662 457990-0, Belgium 32 (0) 2 757 0020, Brazil 55 11 3262 3599, Canada 800 433 3488, China 86 21 5050 9800, Czech Republic 420 224 235 774, Denmark 45 45 76 26 00, Finland 385 (0) 9 725 72511, France 33 (0) 1 48 14 24 24, Germany 49 89 7413130, India 91 80 41190000, Israel 972 3 6393737, Italy 39 02 413091, Japan 81 3 5472 2970, Korea 82 02 3451 3400, Lebanon 961 (0) 1 33 28 28, Malaysia 1800 887710, Mexico 01 800 010 0793, Netherlands 31 (0) 348 433 466, New Zealand 0800 553 322, Norway 47 (0) 66 90 76 60, Poland 48 22 3390150, Portugal 351 210 311 210, Russia 7 495 783 6851, Singapore 1800 226 5886, Slovenia 386 3 425 42 00, South Africa 27 0 11 805 8197, Spain 34 91 640 0085, Sweden 46 (0) 8 587 895 00, Switzerland 41 56 2005151, Taiwan 886 02 2377 2222, Thailand 662 278 6777, Turkey 90 212 279 3031, United Kingdom 44 (0) 1635 523545

For further support information, refer to the *Technical Support and Professional Services* appendix. To comment on National Instruments documentation, refer to the National Instruments Web site at ni.com/info and enter the info code feedback.

Important Information

Warranty

The media on which you receive National Instruments software are warranted not to fail to execute programming instructions, due to defects in materials and workmanship, for a period of 90 days from date of shipment, as evidenced by receipts or other documentation. National Instruments will, at its option, repair or replace software media that do not execute programming instructions if National Instruments receives notice of such defects during the warranty period. National Instruments does not warrant that the operation of the software shall be uninterrupted or error free.

A Return Material Authorization (RMA) number must be obtained from the factory and clearly marked on the outside of the package before any equipment will be accepted for warranty work. National Instruments will pay the shipping costs of returning to the owner parts which are covered by warranty.

National Instruments believes that the information in this document is accurate. The document has been carefully reviewed for technical accuracy. In the event that technical or typographical errors exist, National Instruments reserves the right to make changes to subsequent editions of this document without prior notice to holders of this edition. The reader should consult National Instruments if errors are suspected. In no event shall National Instruments be liable for any damages arising out of or related to this document or the information contained in it.

EXCEPT AS SPECIFIED HEREIN, NATIONAL INSTRUMENTS MAKES NO WARRANTIES, EXPRESS OR IMPLIED, AND SPECIFICALLY DISCLAIMS ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. CUSTOMER'S RIGHT TO RECOVER DAMAGES CAUSED BY FAULT OR NEGLIGENCE ON THE PART OF NATIONAL INSTRUMENTS SHALL BE LIMITED TO THE AMOUNT THERETOFORE PAID BY THE CUSTOMER. NATIONAL INSTRUMENTS WILL NOT BE LIABLE FOR DAMAGES RESULTING FROM LOSS OF DATA, PROFITS, USE OF PRODUCTS, OR INCIDENTAL OR CONSEQUENTIAL DAMAGES, EVEN IF ADVISED OF THE POSSIBILITY THEREOF. This limitation of the liability of National Instruments will apply regardless of the form of action, whether in contract or tort, including negligence. Any action against National Instruments must be brought within one year after the cause of action accrues. National Instruments shall not be liable for any delay in performance due to causes beyond its reasonable control. The warranty provided herein does not cover damages, defects, malfunctions, or service failures caused by owner's failure to follow the National Instruments installation, operation, or maintenance instructions; owner's modification of the product; owner's abuse, misuse, or negligent acts; and power failure or surges, fire, flood, accident, actions of third parties, or other events outside reasonable control.

Copyright

Trademarks

Patents

For patents covering National Instruments products, refer to the appropriate location: **Help»Patents** in your software, the `patents.txt` file on your CD, or `ni.com/patents`.

WARNING REGARDING USE OF NATIONAL INSTRUMENTS PRODUCTS

(1) NATIONAL INSTRUMENTS PRODUCTS ARE NOT DESIGNED WITH COMPONENTS AND TESTING FOR A LEVEL OF RELIABILITY SUITABLE FOR USE IN OR IN CONNECTION WITH SURGICAL IMPLANTS OR AS CRITICAL COMPONENTS IN ANY LIFE SUPPORT SYSTEMS WHOSE FAILURE TO PERFORM CAN REASONABLY BE EXPECTED TO CAUSE SIGNIFICANT INJURY TO A HUMAN.

(2) IN ANY APPLICATION, INCLUDING THE ABOVE, RELIABILITY OF OPERATION OF THE SOFTWARE PRODUCTS CAN BE IMPAIRED BY ADVERSE FACTORS, INCLUDING BUT NOT LIMITED TO FLUCTUATIONS IN ELECTRICAL POWER SUPPLY, COMPUTER HARDWARE MALFUNCTIONS, COMPUTER OPERATING SYSTEM SOFTWARE FITNESS, FITNESS OF COMPILERS AND DEVELOPMENT SOFTWARE USED TO DEVELOP AN APPLICATION, INSTALLATION ERRORS, SOFTWARE AND HARDWARE COMPATIBILITY PROBLEMS, MALFUNCTIONS OR FAILURES OF ELECTRONIC MONITORING OR CONTROL DEVICES, TRANSIENT FAILURES OF ELECTRONIC SYSTEMS (HARDWARE AND/OR SOFTWARE), UNANTICIPATED USES OR MISUSES, OR ERRORS ON THE PART OF THE USER OR APPLICATIONS DESIGNER (ADVERSE FACTORS SUCH AS THESE ARE HEREAFTER COLLECTIVELY TERMED "SYSTEM FAILURES"). ANY APPLICATION WHERE A SYSTEM FAILURE WOULD CREATE A RISK OF HARM TO PROPERTY OR PERSONS (INCLUDING THE RISK OF BODILY INJURY AND DEATH) SHOULD NOT BE RELIANT SOLELY UPON ONE FORM OF ELECTRONIC SYSTEM DUE TO THE RISK OF SYSTEM FAILURE. TO AVOID DAMAGE, INJURY, OR DEATH, THE USER OR APPLICATION DESIGNER MUST TAKE REASONABLY PRUDENT STEPS TO PROTECT AGAINST SYSTEM FAILURES, INCLUDING BUT NOT LIMITED TO BACK-UP OR SHUT DOWN MECHANISMS. BECAUSE EACH END-USER SYSTEM IS CUSTOMIZED AND DIFFERS FROM NATIONAL INSTRUMENTS' TESTING PLATFORMS AND BECAUSE A USER OR APPLICATION DESIGNER MAY USE NATIONAL INSTRUMENTS PRODUCTS IN COMBINATION WITH OTHER PRODUCTS IN A MANNER NOT EVALUATED OR CONTEMPLATED BY NATIONAL INSTRUMENTS, THE USER OR APPLICATION DESIGNER IS ULTIMATELY RESPONSIBLE FOR VERIFYING AND VALIDATING THE SUITABILITY OF NATIONAL INSTRUMENTS PRODUCTS WHENEVER NATIONAL INSTRUMENTS PRODUCTS ARE INCORPORATED IN A SYSTEM OR APPLICATION, INCLUDING, WITHOUT LIMITATION, THE APPROPRIATE DESIGN, PROCESS AND SAFETY LEVEL OF SUCH SYSTEM OR APPLICATION.

Conventions

The following conventions are used in this manual:

<>	Angle brackets that contain numbers separated by an ellipsis represent a range of values associated with a bit or signal name—for example, DIO<3..0>.
»	The » symbol leads you through nested menu items and dialog box options to a final action. The sequence **File»Page Setup»Options** directs you to pull down the **File** menu, select the **Page Setup** item, and select **Options** from the last dialog box.
	This icon denotes a note, which alerts you to important information.
	This icon denotes a caution, which advises you of precautions to take to avoid injury, data loss, or a system crash.
bold	Bold text denotes items that you must select or click in the software, such as menu items and dialog box options. Bold text also denotes parameter names.
italic	Italic text denotes variables, emphasis, a cross-reference, or an introduction to a key concept. Italic text also denotes text that is a placeholder for a word or value that you must supply.
`monospace`	Text in this font denotes text or characters that you should enter from the keyboard, sections of code, programming examples, and syntax examples. This font is also used for the proper names of disk drives, paths, directories, programs, subprograms, subroutines, device names, functions, operations, variables, filenames, and extensions.
`monospace bold`	Bold text in this font denotes the messages and responses that the computer automatically prints to the screen. This font also emphasizes lines of code that are different from the other examples.
`monospace italic`	Italic text in this font denotes text that is a placeholder for a word or value that you must supply.

Contents

Chapter 1
Introduction

Chapter 2
C Language Reference

Chapter 3
Ada Language Reference

Chapter 4
Generating Code for Real-Time Operating Systems

Chapter 5
Generated Code Architecture

Chapter 6
Vectorized Code Generation

Chapter 7
Code Optimization

Chapter 8
AutoCode Sim Cdelay Scheduler

Chapter 9
Global Scope Signals and Parameterless Procedures

1 Introduction

This manual provides reference material for using AutoCode to write production quality code using graphical tools. Together with the *AutoCode User Guide* and the *Template Programming Language User Guide*, AutoCode documentation describes how to generate robust, high-quality, real-time C or Ada source code from SystemBuild block diagrams.

Manual Organization

This manual includes the following chapters:

- Chapter 1, *Introduction*, provides an overview of the rapid prototyping concept, the automatic code generation process, and the nature of real-time generated code.
- Chapter 2, *C Language Reference*, discusses files used to interface AutoCode and the generated C code to your specific platform and target processor, and target-specific utilities needed for simulation and testing.
- Chapter 3, *Ada Language Reference*, discusses files used to interface AutoCode and the generated Ada code to your specific platform and target processor, and target-specific utilities needed for simulation and testing.
- Chapter 4, *Generating Code for Real-Time Operating Systems*, describes the RTOS configuration file and functionality provided for generating code for real-time operating systems.
- Chapter 5, *Generated Code Architecture*, supplies more details about the content and framework of the generated code. This includes storage usage, various procedures, specialized blocks, and subsystems.
- Chapter 6, *Vectorized Code Generation*, discusses various ways to generate vectorized code. This includes describing the options available, design guidelines, and implementation details about the vectorized code.
- Chapter 7, *Code Optimization*, discusses how to optimize the generated code. This includes explaining the details of generating production quality code for micro controller-based applications.

- Chapter 8, *AutoCode Sim Cdelay Scheduler*, discusses the Sim Cdelay low-latency scheduler.
- Chapter 9, *Global Scope Signals and Parameterless Procedures*, discusses additional signals and procedures.

This guide also has an *Index*.

General Information

As an integral part of the rapid prototyping concept, AutoCode lets you generate high-level language code from a SystemBuild block diagram model quickly, automatically, and without programming skills. The *AutoCode User Guide* describes the processes for generating code, including the parameters that you must use. This manual provides details of how AutoCode actually works, so that you will have an idea of what to expect from AutoCode if you attempt to modify the generation of code.

Configuration File

The configuration file is a text file containing tables of information related to the generated source code components of a model, like subsystems and nonscheduled procedure SuperBlocks. Each table contains configuration information about its respective component. Various configuration files are supplied with AutoCode, including one for Real-Time Operating Systems (RTOS) as described in Chapter 4, *Generating Code for Real-Time Operating Systems*. For additional configuration information, refer to the *AutoCode User Guide*.

Language-Specific Information

This manual describes some of the details of AutoCode operation for both the C and Ada languages.

Specific topics include the following:

- Stand-alone (`sa`) files
- Fixed-point code generation
- UserCode Blocks (UCBs)
- Macro Procedure Blocks
- Procedure SuperBlocks

Structure and Content of the Generated Code

This reference includes detailed descriptions about what is generated for many of the blocks used within a model. Also, the framework of the generated code is discussed to show how all of the pieces work together to form an executable simulation. This discussion is only relevant to those designers who are either writing their own templates or who are striving to optimize the generated code.

Topics include the following:

- Generating code for use within real-time operating systems
- Code architecture
- Vectorized code generation

Using MATRIXx Help

MATRIXx 7.*x* provides a hypertext markup language (HTML) help system. The *MATRIXx Help* is a self-contained system with multiple hypertext links from one component to another. This help, augmented by online and printed manuals, covers most MATRIXx topics except for installation.

The *MATRIXx Help* runs on Netscape. The MATRIXx CD-ROM includes the supported version. On UNIX systems, an OEM version of Navigator is automatically included in the MATRIXx installation. On PCs, Netscape must be installed independently using the Netscape installation procedure included on the MATRIXx CD.

Additional Netscape Information

For more information on Netscape products, visit the Netscape Web site at `http://home.netscape.com`.

Related Publications

National Instruments provides a complete library of publications to support its products. In addition to this guide, publications that you may find particularly useful when using AutoCode include the following:

- *AutoCode User Guide*
- *Template Programming Language User Guide*
- *Xmath User Guide*
- *SystemBuild User Guide*
- *BlockScript User Guide*
- *DocumentIt User Guide*

For additional documentation, refer to the *MATRIXx Help* or visit the National Instruments Web site at `ni.com/manuals`.

2

C Language Reference

This chapter discusses files used to interface AutoCode and the generated C code to your specific platform and target processor. This chapter also describes target-specific utilities needed for simulation and testing.

Stand-Alone Simulation

The template provided for C code generation produces code that, when compiled and linked with stand-alone files, forms a stand-alone simulation. This simulation can be executed with MATRIXx-style data as input, and produces results that can be loaded back into Xmath for analysis. You must compile the generated code along with the stand-alone library to produce the simulation executable.

Chapter 2, *Using AutoCode*, of the *AutoCode User Guide* describes how to compile the code and stand-alone library, generate sample input data, and load the data into Xmath for analysis.

Compiling on Various Supported Platforms

The generated code usually includes platform-specific code. Most of this code is not generated by AutoCode; rather, that code exists in the template. Also, the stand-alone library has platform-specific code to deal with file I/O and floating-point numerics. You must compile the generated code and the stand-alone library with a defined preprocessor symbol appropriate for your platform (refer to Table 2-1). For example, on the Solaris platform, the compile statement is similar to:

```
% acc -DSOLARIS -o simulation simmodel.c sa_*.o -lm
```

Note This example assumes the stand-alone library was compiled into separate object files into the current working directory and that the stand-alone header files also exist in the current working directory.

Table 2-1. Recognized C Preprocessor Defines for Supported Platforms

Platform	Preprocessor Define	Compiler Switch
AIX (IBM UNIX)	IBM	`-DIBM`
Compaq Tru64 5.0	OSF1	`-DOSF1`
HPUX (700 series)	HP700	`-DHP700`
HPUX (other than 700)	HP	`-DHP`
SGI IRIX	SGI	`-DSGI`
Sun Solaris 2.*x*	SOLARIS	`-DSOLARIS`
Windows 2000/NT/9*x*	MSWIN32	`-DMSWIN32`

Stand-Alone Library

This section describes the system-specific and target-specific stand-alone (`sa`) files supplied with your system.

System-Specific Files

National Instruments furnishes files to interface AutoCode and the generated code to your development platform (AIX, Compaq, HP, SGI, Solaris, Windows) and to the target processor in your test bed or rapid-prototyping system. Both header (`.h` extension) and C (`.c` extension) source files are provided in your `src` distribution directory. The names of the distribution directories and files are shown in Table 2-2.

Table 2-2. Distribution Directories and Files

Platform		UNIX	Windows
Top-Level Directory	Environment variable:	`$MTXHOME`	`%MTXHOME%`
Executables	Directory: Executable:	`$MTXHOME/bin` `autostar`	`%MTXHOME%\bin` `autostar`

Utilities	Directory:
	Files:
	Script:
		`$CASE/ACC/src`
		`sa_*.c, sa_*.h`
		`compile_c_sa.sh`
			`%CASE%\ACC\src`
			`sa_*.c, sa_*.h`
			`compile_c_sa.bat`

Table 2-2. Distribution Directories and Files (Continued)

Platform		UNIX	Windows

Templates	Directory:
	Templates:
	Direct Access Templates:
		`$CASE/ACC/templates`
		`c_sim.tpl, c_intgr.tpl`
		`c_sim.dac`
			`%CASE%\ACC\templates`
			`c_sim.tpl, c_intgr.tpl`
			`c_sim.dac`

Demos	Directory:	`$XMATH/demos`	`%XMATH%\demos`

- The principal file is `sa_utils.c`, the stand-alone utilities file. At the time that you compile `sa_utils.c` and your generated code program, you must make the following header files available locally:

  `sa_sys.h` `sa_defn.h` `sa_utils.h`
  `sa_math.h` `sa_types.h` `sa_intgr.h`
  `sa_math.c` (C file from distribution directory)

- If the generated code contains time references (that is, if the variable `RT_DURATION` appears in the generated code), file `sa_time.h` must be available in the local directory.
- If you have defined any UserCode Blocks, the following files must be available locally:

  `sa_user.c` AutoCode UCB template
  `sa_user.h` Header file

Note If you use the fuzzy logic block in your model, the `sa_fuzz.c` and `sa_fuzzy.h` files must be available locally. Use this file only when linking an AutoCode UCB back into SystemBuild. Also, `sa_user.c` is just a template for a UCB and only needs to be compiled into the stand-alone simulation if your SystemBuild model includes UCBs. Refer to the *Linking Handwritten UCBs (for AutoCode) with SystemBuild* section for more information.

- If you use fixed-point data types in your model, files that implement fixed-point operations must be available. For more details about AutoCode fixed-point implementation and fixed-point support files, refer to the *C Fixed-Point Arithmetic* section.

Table 2-3 summarizes the most significant header files. The `sa_` prefix indicates stand-alone.

Table 2-3. Header Files

File	Purpose
sa_sys.h	Defines the development platform. Contains a C preprocessor #define statement for each supported platform.
sa_types.h	Defines the supported data types and certain math constants.
sa_defn.h	Defines constants for generated code, error codes, and mapping for ANSI features such as const and volatile.
sa_intgr.h	Contains definitions of integration algorithms (including user-supplied integrator) used in code generation of hybrid or continuous systems.
sa_fuzzy.h	Contains definitions of fuzzy logic support routines.
sa_utils.h	Contains external function prototypes for the stand-alone utilities.
sa_math.h	Declares certain extensions to ANSI-Standard C math functions. The sa_math.c file, which contains the code for the extensions, is also required.
sa_time.h	Declares a time-related variable.
sa_user.h	Furnishes a function prototype for UCBs.

Data Types

Several of the target-specific utilities are involved with data types (in the sa_types.h file). The three following data types are defined for the C Code Generator:

RT_FLOAT — Corresponds to C type double or float, depending on your C compiler.

RT_INTEGER — Corresponds to C type integer.

RT_BOOLEAN — Corresponds to C type integer.

At compilation, you must make available the sa_types.h header file, which declares these types. This file is in the src distribution directory on your system; you can edit a copy of it as required. The structure STATUS_RECORD also is declared in sa_types.h to be used with hand-coded UserCode Blocks. You can modify sa_types.h if you need

to. For example, `RT_INTEGER` can be redefined as `long int` if arithmetic overflow becomes a problem on a given platform.

Target-Specific Utilities

Target-specific utilities (in `sa_utils.c`) perform hardware, application, and C-specific tasks that are required for simulation and testing. They can be modified to support the generated code on different target computers. As furnished, these routines simulate I/O operations that would occur in a simulation environment, using input files created using MATRIXx. These files are intended to remain unmodified for use in comparing your simulations with generated code outputs. However, for target-system usage on your rapid prototyping or end-user system, these routines can be modified or rewritten completely and recompiled for the target system. When you do this, be sure to keep a copy of the `sa_utils.c` file or keep separate versions of the files in separate directories.

There is no requirement that the file be named `sa_utils.c`; however, the name you use must be specified at link time. Inside the file, the names of the external variables, functions, and other references must be preserved. As furnished for this release, the routines are written in C, but this is not required. If you rewrite the routines, they should still be written in a language that offers direct control of the I/O and interrupt hardware of the target computer and can be linked to the object form of the generated C program. Normally, these utilities need to be created only once for each target computer. In general, a given set of target-specific utilities need only be changed when the target hardware is modified. The routines are shown in Table 2-4.

Table 2-4. Target-Specific Utility Routines

Routine	Description
`Enable( )`	Unmask timer interrupt.
`Disable( )`	Mask timer interrupt.
`Background( )`	Background polling loop.
`Error( ), fatalerr( )`	Error handlers.
`Implementation_Initialize( )`	Initialize I/O hardware and perform other implementation-specific initialization tasks.

Table 2-4. Target-Specific Utility Routines (Continued)

Routine	Description
`Implementation_Terminate( )`	Perform implementation-specific termination tasks.
`External_Input( )`	Collect external inputs.
`External_Output( )`	Post external outputs.
`Signal_Remote_Dispatch( )`	Multiprocessor implementations only; signal secondary processors that a dispatch table is available.

The `sa_utils.c` file contains comments about each of the routines as they are used for comparing simulation with generated code results.

After you generate code, link the generated code with `sa_*.o` object files (refer to Chapter 2, *Using AutoCode*, of the *AutoCode User Guide)*. For example, on UNIX platforms:

```
cc -o gen_ap
gen_ap.c $ISI/ACC/src/sa_*.o -l other libraries
```

where *gen_ap* is the name of the generated code file.

enable( ), disable( ), and background( ) Functions

`enable( )` unmasks timer interrupts. `disable( )` masks timer interrupts to prevent re-entry of the scheduler during critical sections. `enable( )` and `disable( )` are not needed in some implementations. These functions are furnished as stubs and `#defined` as `NULL` in the `sa_utils.h` file.

The `background( )` function, as provided in `sa_utils.c`, merely invokes the scheduler for the appropriate number of computation cycles and calls the error handler if any scheduler error occurs.

error( ) and fatalerr( ) Functions

```
void fatalerr(RT_INTEGER ERROR);
```

```
void error(RT_INTEGER NTSK,RT_INTEGER ERRORFLAG);
```

Two error functions are provided, `fatalerr( )` and `error( )`. The `fatalerr( )` function reports exception conditions detected by the functions in the `sa_utils.c` file. `error( )` reports conditions detected by the generated code during execution. Not all reported conditions are errors. These functions can be invoked for deactivating all necessary functions and then passing an alarm to the external environment or for initiating recovery action. You can choose either to return from an error function or to halt the machine.

fatalerr( ): Stand-Alone Utilities Detected Errors

Several error conditions are trapped in `sa_utils.c` by default, but you can expand this capability in your own versions of the program, detecting your own error conditions and adding your own messages. The `ERROR` value that is returned is evaluated by a C language switch-case statement. Any `RT_INTEGER` value can be used for an error indication, except that the value of `-1` is reserved for use in the scheduler.

The following are generated messages displayed in the default version of the `sa_utils.c` file. Most of these messages pertain to the processing of the input and output files for execution of generated code.

INPUT FILE IS NOT IN Xmath {matrixx,ascii} FORMAT

Save the file in MATRIXx ASCII format from Xmath, then try again.

INPUT FILE IS NOT V7.0 OR LATER

The input data file was generated using an obsolete version of MATRIXx. Save the file in MATRIXx ASCII format from Xmath, then try again.

INPUT FILE CONTAINS MORE THAN TWO ARRAYS

INPUT TIME VECTOR NOT ONE COLUMN

INPUT U DIMENSION NOT (TIME x NUMBER OF INPUTS)

The following messages indicate a bad input file.

INPUT TIME VECTOR TOO LARGE

INPUT U ARRAY TOO LARGE FOR AVAILABLE STORAGE

OUTPUT STORAGE EXCEEDS THE AVAILABLE STORAGE

The following messages indicate that the size of the input file has exceeded one or more of the storage allocation size limits established by `sa_utils.c`. These limits are `#defined` at the very beginning of the `sa_utils.c` header, just after the `#include` header statements. Refer to

the comments there and adjust the limits accordingly, then recompile and relink the `sa_utils.c` file.

ERROR OPENING THE INPUT FILE

ERROR OPENING THE OUTPUT FILE

A problem was encountered opening the input or output file. Possible causes include a file protection violation.

UNKNOWN ERROR

A value of the `ERROR` variable occurred that was not one of those defined for the switch-case statement. Check any error indications you may have introduced.

ERROR: Conditions Detected in the Generated Code

The `RT_INTEGER` variable `ERROR_FLAG` is passed if an error occurs in running the generated code. The following conditions are trapped, not all of which indicate that an error has occurred.

The following messages might be generated during the execution of the generated code:

Stop Block encountered in task *n*

This is not necessarily an error. This refers to a SystemBuild Stop Simulation Block encountered in the execution of the generated code.

Math Error encountered in task *n*

Check your model for overflows, division by zero, and similar problems.

User code error encountered in task *n*

Refer to Chapter 15, *UserCode Blocks*, of the *SystemBuild User Guide* or the source listing of the USR01 routine for meanings of UCB errors. You can extend the scope of these messages, so it might be one of yours.

Unknown error encountered in task *n*

A possible cause is an incorrect user-written error condition in the generated code.

Time overflow occurred in task *n*

This indicates a subsystem (or task) overflow. Refer to the *Scheduler Errors* section of Chapter 4, *Managing and Scheduling Applications*, of the *AutoCode User Guide* for timing overflow conditions.

Implementation_Initialize( ) Function

```
void Implementation_Initialize (RT_FLOAT *BUS_IN
RT_INTEGER, NI, RT_FLOAT BUS_OUT, RT_INTEGER NO,
RT_FLOAT SCHEDULER_FREQ);
```

In the default version of `sa_utils.c` (simulation comparison), this function initializes the I/O for the system by loading input data from the user-furnished MATRIXx `FSAVE` input file. In the version of this routine that you write to make the generated code work with the target computer, this routine performs implementation-specific initialization processes for the real-time system. These might include, but are not limited to, the following:

- Initialize the interrupt system of the target computer.
- Initialize the math coprocessor, if any.
- Set up shared memory and other hardware-dependent processes.
- Initialize I/O hardware.
- Initialize the clock-timer of the target system to request interrupts at the minor cycle of the control system; that is, the time period required between interrupts for synchronous operation of the tasks, as calculated by AutoCode from the block diagrams.

Implementation_Terminate( ) Function

```
void Implementation_Terminate(void)
```

In the default version of `sa_utils.c` (simulation comparison), this function completes the I/O processing for the execution of the system by writing output data to the output file and closing the file. In the version of this routine that you write to make the generated code work with the target computer, this routine might be called on to perform many kinds of implementation-specific shutdown processes for the real-time system in addition to data completion tasks. These might include, but are not limited to, the following:

- Disable interrupt hardware.
- Free up shared memory and other resources.
- De-initialize I/O hardware.

External_Input ( ) Function

```
RT_INTEGER External_Input(void)
```

`External_Input( )` is for use in the simulation comparison mode; it reads in external input data from your specified `FSAVE` input file. The data appears in `XINPUT`, an array of type `RT_FLOAT`, dimensioned equal to the input vector (`T`- and `U`-vectors) defined at simulation time. No data conversion is required in this version of the generated code, because all data is passed as arrays of type `RT_FLOAT`. The routine returns the value of `SCHEDULER_STATUS`, which reports on the success of the input operation and is passed to the `External_Input( )` routine by the scheduler. In the target version of `sa_utils.c`, the operation of this function is much the same; every time it is called, `External_Input( )` returns an input vector from the hardware sensors.

External_Output ( ) Function

```
RT_INTEGER External_Output(void)
```

`External_Output( )` is for use in the simulation comparison mode; it posts external output data to your specified output file. The data is presented in `XOUTPUT`, an array of type `RT_FLOAT`, dimensioned equal to the output `Y` vector as defined at simulation time. No data conversion is required in this version of the generated code, because all data is passed as arrays of type `RT_FLOAT`. The routine returns the value of `SCHEDULER_STATUS`, which is passed to it by the scheduler. In the target version of `sa_utils.c`, the operation of this function is much the same; every time it is called, `External_Output( )` posts an output vector to the software data bus.

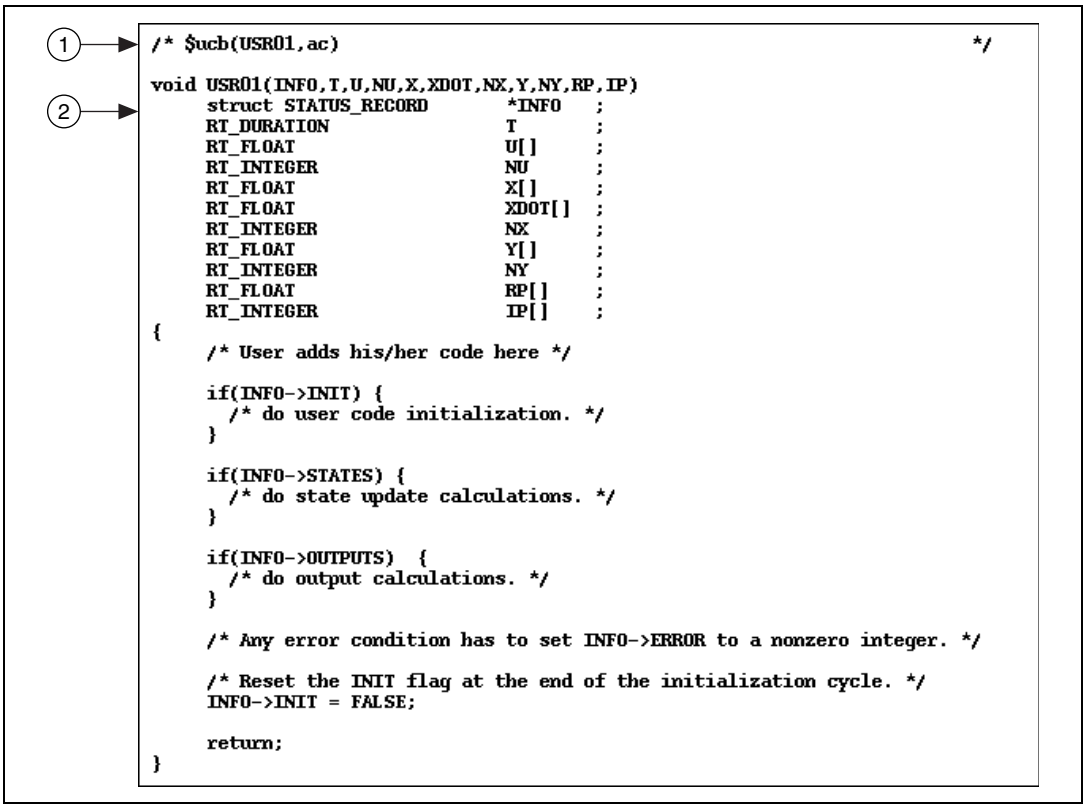
UserCode Blocks

This section describes how to link UserCode Blocks (UCBs) with AutoCode or SystemBuild applications. AutoCode supports all SystemBuild data types for inputs and outputs and the generation of the fixed and variable interface options. The variable interface does not include optional arguments to the user code—for example, states—if that argument is not specified in the UCB. Unless otherwise stated, the following sections describe the fixed interface option of a UCB.

Linking Handwritten UCBs with AutoCode Applications

To write code for UserCode Blocks (UCBs), refer to the `sa_user.c` file, which is provided in your `src` distribution directory.

The `sa_user.c` file contains an example of a UCB function declaration (refer to the *Implementing Handwritten UCBs* section). If your model has more than one UCB, each prototype must have a unique name. Make one or more copies of this file and insert the code that implements the algorithm(s) of the UCB(s). One or more uniquely-named UCB code functions can be placed inside each copy of the `sa_user.c` file and the copies can be given any convenient names. If renamed files are included, the names can be placed into the stand-alone file compilation script (`compile_c_sa.sh`) for automatic compilation. If the UCB function—for example, USR01, refer to callout 2 of Figure 2-1—is renamed, all other occurrences of USR01 in `sa_user.c` and `sa_user.h` also must be changed appropriately, including the occurrence in directive `$ucb`. Refer to callout 1 of Figure 2-1.




```
/* $ucb(USR01,ac)                                                    */

void USR01(INFO,T,U,NU,X,XDOT,NX,Y,NY,RP,IP)
     struct STATUS_RECORD       *INFO   ;
     RT_DURATION                T       ;
     RT_FLOAT                   U[]     ;
     RT_INTEGER                 NU      ;
     RT_FLOAT                   X[]     ;
     RT_FLOAT                   XDOT[]  ;
     RT_INTEGER                 NX      ;
     RT_FLOAT                   Y[]     ;
     RT_INTEGER                 NY      ;
     RT_FLOAT                   RP[]    ;
     RT_INTEGER                 IP[]    ;
{
     /* User adds his/her code here */

     if(INFO->INIT) {
       /* do user code initialization. */
     }

     if(INFO->STATES) {
       /* do state update calculations. */
     }

     if(INFO->OUTPUTS)  {
       /* do output calculations. */
     }

     /* Any error condition has to set INFO->ERROR to a nonzero integer. */

     /* Reset the INIT flag at the end of the initialization cycle. */
     INFO->INIT = FALSE;

     return;
}
```

Figure 2-1. Example UserCode Function File (sa_user.c)

The `$ucb` directive is recognized and interpreted by the automatic linking facility of the simulator in SystemBuild to distinguish between UCBs written for simulation purposes using SystemBuild only and UCBs written for linking with AutoCode applications. The exact name of the UCB function must be specified in the `$ucb` directive if this UCB function is used for simulation using SystemBuild.

The computations performed by UCBs can update both block states and outputs. Execution of each section is controlled by flags set by the simulator in the INFO structure. Refer to the *Implementing Handwritten UCBs* section.

Implementing Handwritten UCBs

Arguments are passed for each call to the UCB in the following order:

`INFO, T, U, NU, X, XDOT, NX, Y, NY, R_P, and I_P`

Pointers to all of the arrays (`U`, `X`, `XD`, `Y`, `R_P`, `I_P`) and scalers corresponding to array sizes (`NU`, `NX`, `NY`) are passed to each call to the UCB. The sizes of `R_P` and `I_P` arrays must be entered into the UCB dialog to ensure that proper storage is allocated by the caller. Also, the initial conditions for states have to be specified in the dialog. Table 2-5 lists the type and purpose of UCB arguments used in the fixed calling method.

Table 2-5. UCB Calling Arguments and Data Types for the Fixed Interface

Argument	Data Type	Description
INFO	struct STATUS_RECORD*	A pointer to STATUS_RECORD structure representing operation requests and status.
T	RT_DURATION	Elapsed time.
U[ ]	RT_FLOAT	An array (of size NU) of inputs to the UCB.
NU	RT_INTEGER	The number of inputs to the UCB.
X[ ]	RT_FLOAT	An array (of size NX) of state variables of the UCB.
XDOT[ ]	RT_FLOAT	An array (also of size NX). Includes the next discrete states of X for the discrete subsystems, and the derivative of X for the continuous subsystems.
NX	RT_INTEGER	The number of states (and next states).
Y[ ]	RT_FLOAT	An array (of size NY) of outputs from the UCB.
NY	RT_INTEGER	The number of outputs from the UCB.
R_P[ ]	RT_FLOAT	An array of real parameters.
I_P[ ]	RT_INTEGER	An array of integer parameters.

The operations within UCBs are controlled by the argument INFO, a pointer to a structure of type STATUS_RECORD that is passed as part of the argument list for each UCB (located in sa_types):

```
typedef struct STATUS_RECORD
{
   RT_INTEGER ERROR;
   RT_BOOLEAN INIT;
   RT_BOOLEAN STATES;
   RT_BOOLEAN OUTPUT;
}  RT_STATUS_RECORD;
```

The following example shows the general form of UCB equations for AutoCode and indicates how the INFO status record pointer is used to control the computations.

```
if (INFO->INIT) {
  /* do user code initialization */
  INFO->INIT = 0;
}
if (INFO->OUTPUTS) {
  /* do output calculations having the general form:
     Y = H(T,X,XD,U,RP,IP); */
}
if (INFO->STATES) {
  /* do state update calculation with the general form:
     XD = F(T,X,XD,U,RP,IP); */
}
```

When an error occurs within the UCB, set INFO->ERROR equal to some nonzero integer value as an error return. Also, make sure that INFO->INIT is set to FALSE (0) at the end of the initialization cycle.

The process of linking handwritten UCBs (in usr_dsp.c) with an AutoCode application (in dsp.c) is depicted in Figure 2-2. The SystemBuild model in Figure 2-2 depicts the filter described by the difference equation:

$$y(k) = -1/2 * y(k - 2) + 3 * u(k) + u(k - 2)$$

The UserCode Block provides the interface to the handwritten function usr_dsp( ) implemented in the file usr_dsp.c. The usr_dsp( ) function essentially implements the behavior of the same filter.

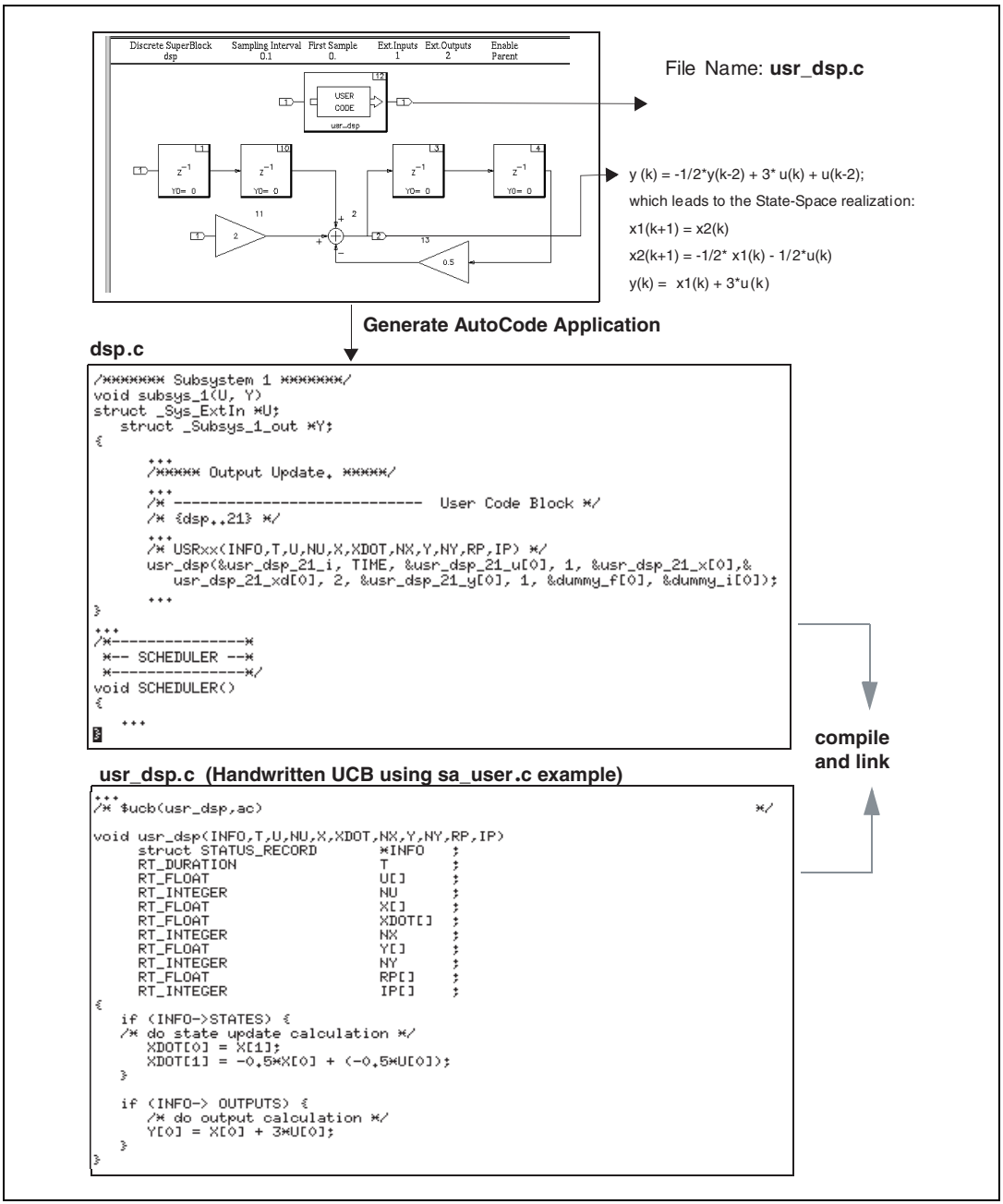


Figure 2-2. Linking Handwritten UCBs with AutoCode Applications

Linking Handwritten UCBs (for AutoCode) with SystemBuild

After you have written a UCB to create an AutoCode application, you can use the same UCB for simulation. SystemBuild can automatically compile and link your UserCode function into the simulation engine (release 4.0 and later). Refer to Figure 2-3.

SystemBuild provides the `usr01.c` example file and AutoCode provides the `sa_user.c` example file for writing UCBs. These files are different and should not be used interchangeably for linking UCBs with SystemBuild and AutoCode applications. `usr01.c` is strictly meant to link user-written code with the SystemBuild simulator. `sa_user.c` is meant to link user-written code with AutoCode applications. After the code has been written using `sa_user.c`, the same file might be linked with SystemBuild. The `$ucb` directive in `usr_dsp.c` in Figure 2-3 makes it possible to link handwritten UCBs for AutoCode with SystemBuild. However, code written using `usr01.c`, provided by SystemBuild, cannot be linked with an AutoCode application.

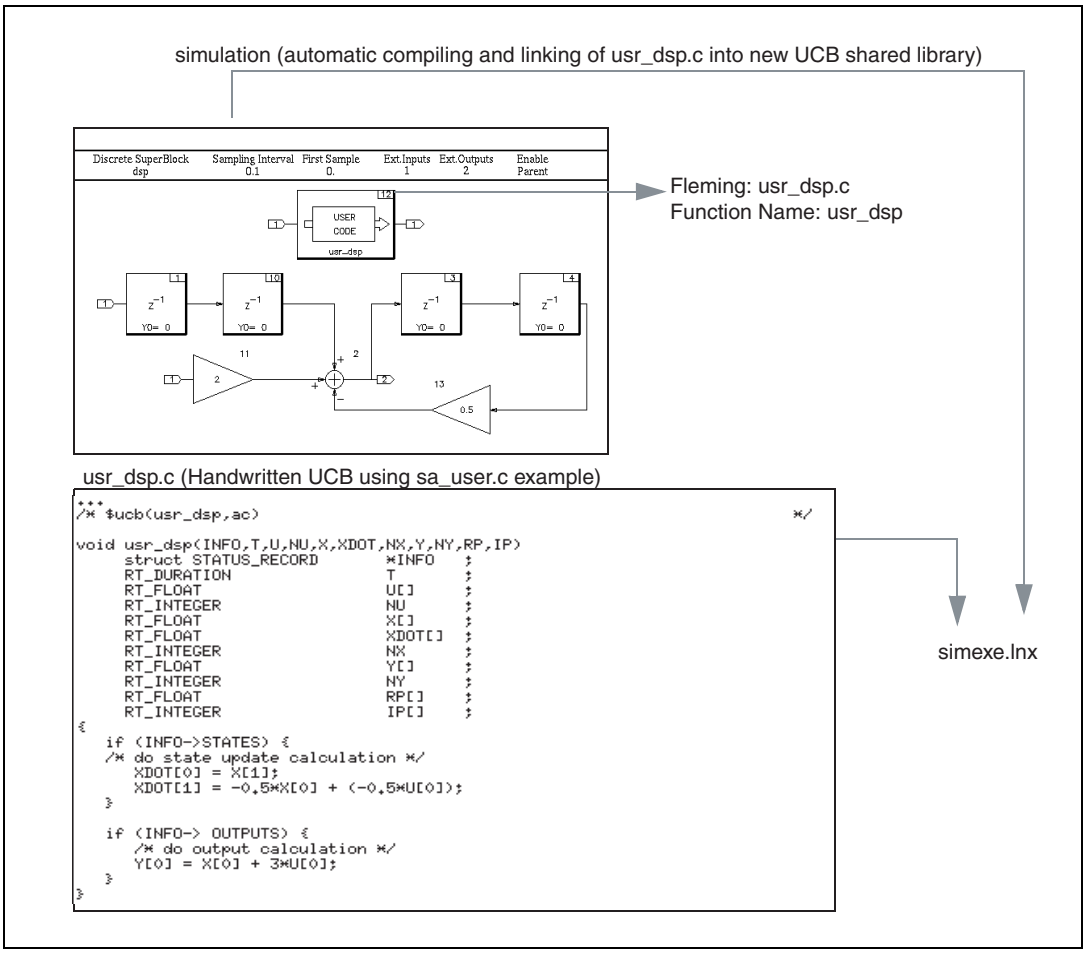


Figure 2-3. Linking Handwritten UCBs with the SystemBuild Simulator

The arguments to a UCB written only for linking with the SystemBuild simulator (using `usr01.c`) are inherently different than the arguments to a UCB written for linking with an AutoCode application (using `sa_user.c`). This difference stems from several sources. For example, UCBs written for linking with an AutoCode application might not take advantage of some features supported in the SystemBuild simulator, such as linearization. Linearization is not supported in AutoCode generated code. Also, highly optimized implementation of a UCB tends to be more of an issue when linking it with AutoCode generated code. If you have handwritten code for simulation with SystemBuild using the UCB format in `usr01.c` and plan to use the same algorithm with AutoCode-generated

applications, make sure you adapt the same algorithm in the body of a function using the AutoCode UCB arguments as in `sa_user.c`.

Variable Interface UCB

The preceding sections described the fixed interface; however, a UCB can also use the variable interface option. For information on how to specify the variable interface option, refer to the *SystemBuild User Guide*. When a model containing a variable interface UCB is generated, the code to call the UCB varies depending upon the data types and optional arguments specified by the UCB. For example, if the UCB does not have any states, arguments related to states are not passed. The following sections describe what AutoCode generates for the variable interface.

Interface Ordering

The variable interface consists of arguments required to meet the specification of the UCB. These arguments are passed relative to a fixed total ordering of all possible arguments. The following is the total ordering. Refer to Table 2-5 for descriptions of the arguments:

```
INFO, T, {u1, u2, ...}, {y1, y2, ...}, {X[], XD[], NX},
{R_P, NRP}, {I_P, NIP}
```

Note Arguments shown within braces `{}` are related; either all or none of these arguments will be generated.

Interface Examples

The following samples of generated code are based on the specification of a UCB.

```
/* INFO not used; T used; 2 inputs; 1 output; */
usr01(TIME, in1, in2, &out1);

/* INFO not used; T not used; 1 input; 5IPs */
usr02(in1, &IP[0], 5);

/* INFO used; T not used; 1 output; 4 states */
usr03(&info, &out1, &X[0], &XD[0], 4);
```

Inputs and Outputs

The fixed interface requires that all inputs and outputs be the floating-point (RT_FLOAT) data type. However, the variable interface supports all data types for the inputs and outputs of the UCB. Consequently, this aspect of the variable interface can lead to errors.

As previously stated, the inputs and outputs of the UCB will have the same data type as specified in the model diagram. Inputs are passed by value while outputs are passed by reference. In addition, the “shape” of an argument describes whether it is a scalar value or an array. For information on how to specify the data type and shape of the inputs and outputs of the UCB, refer to the *SystemBuild User Guide*.

Another complexity relates to vectorization with AutoCode code generation. By design, the variable interface UCB is insulated from the many possible combinations. The interface to the UCB will remain constant relative to all of the AutoCode optimization. As a result, AutoCode performs any needed copying—for example, staging—of inputs and outputs to make sure the proper arguments are passed to the UCB. For example, if the UCB expects an array of five inputs, yet scalar code is generated, AutoCode creates a temporary array for those inputs and passes that temporary array to the UCB.

Function Prototype

National Instruments recommends that you create a function prototype for your variable interface UCB functions. This will provide some additional checking that the compiler can do to ensure that the generated interface matches what you expected without your implementation of the UCB function.

The following is a sample prototype for a variable interface UCB function:

```
void ucb01(
             RT_FLOAT    change[2],     /*** inputs ***/
              RT_INTEGER  xpos,
              RT_INTEGER  ypos,

             RT_FLOAT    *rate,         /*** outputs ***/
              RT_INTEGER  posdata[3],

              RT_FLOAT    *RP,           /*** RP ***/
              RT_INTEGER  NRP
);
```

Linking a Variable Interface UCB with the Simulator

Unlike the fixed interface which provides an automatic method for linking with the Simulator, the variable interface is too complicated for that method. As a result, you are required to create a "wrapper" function that interfaces between the Simulator and your code. For information on how to create the wrapper for the Simulator, refer to the *SystemBuild User Guide*.

Procedure SuperBlocks

This section describes how to generate and link Procedure SuperBlocks.

Generating Reusable Procedures

Generate a reusable procedure from your Procedure SuperBlock as described in Chapter 3, *Ada Language Reference*. Refer to callout 1 of Figure 2-4. Along with the algorithmic procedure (refer to callout 2), AutoCode also generates the respective hook procedure or wrapper (refer to callout 3) for automatic linking with the SystemBuild simulator.

Refer to Chapter 5, *Generated Code Architecture*, and Chapter 9, *Global Scope Signals and Parameterless Procedures*, for more information about generating procedures.

Linking Procedures with the SystemBuild Simulator

Replace the procedure SuperBlock with a UserCode Block (UCB), refer to callout 4 of Figure 2-4 and place the appropriate file name and function name in the UCB dialog box entries. The function name should be that of the Procedure SuperBlock with `_ucbhook` appended to it.

When simulating the model, the SystemBuild simulator automatically compiles and links the generated code, and executes the simulation with the new code library created at this point.

If more than one Procedure SuperBlock resides in the top-level discrete SuperBlock from which the procedures are generated, AutoCode will only generate a UCB wrapper for the first Procedure SuperBlock it encounters. If you want to generate wrappers for the other Procedure SuperBlocks, place each Procedure SuperBlock in a separate Discrete SuperBlock and generate code. You can also modify the template file to generate wrappers for all procedures in your model by invoking the TPL function `proc_ucb_hook` on all Procedure SuperBlocks. Notice how default template file `c_sim.tpl` invokes `proc_ucb_hook` for one Procedure SuperBlock, modify it to loop on all Procedure SuperBlocks, and invoke

`proc_ucb_hook`. Refer to the *Template Programming Language User Guide*.

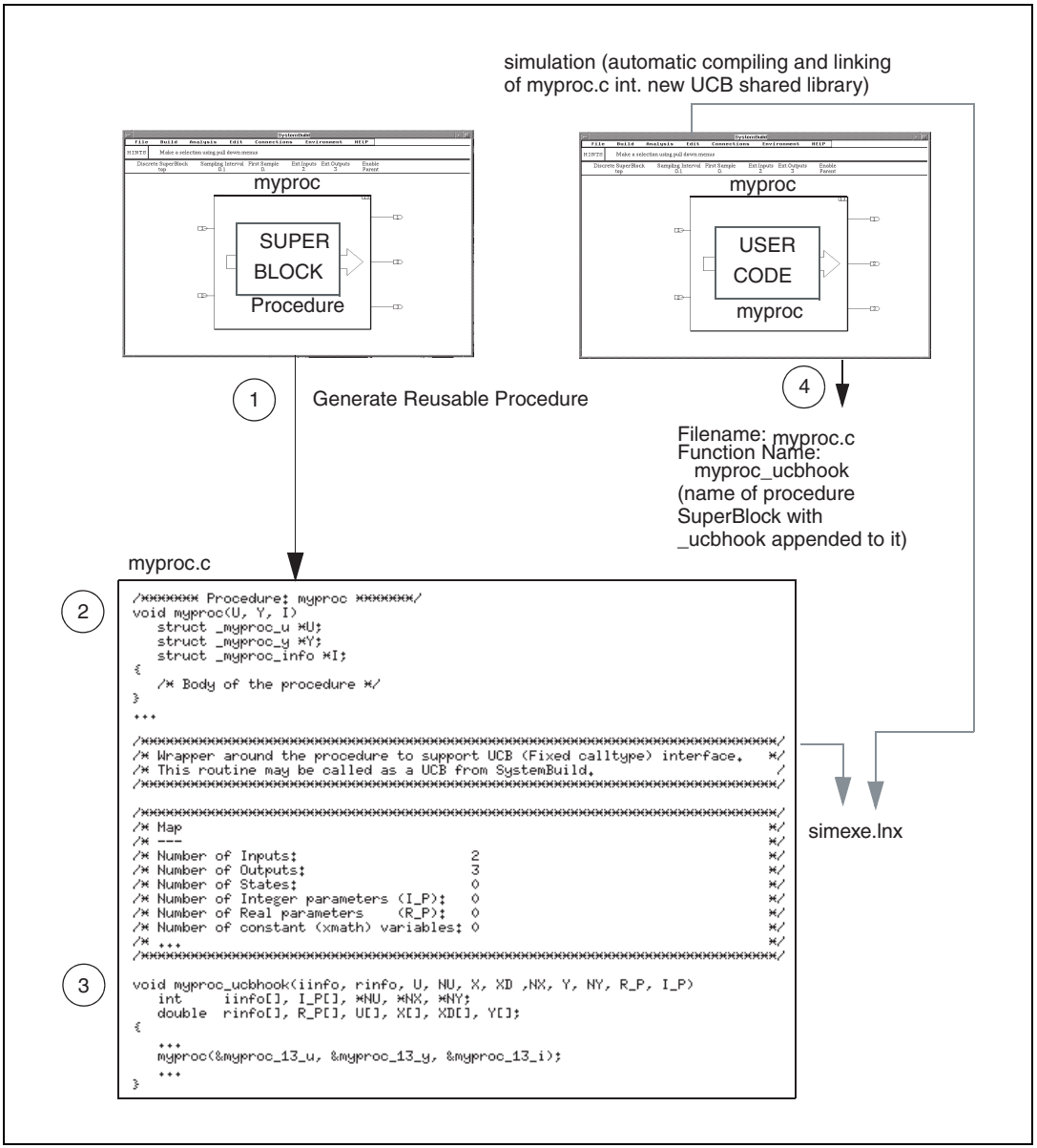


Figure 2-4. Linking Generated Reusable Procedures

Linking Procedures with Real-Time Applications or Simulator

Generate reusable procedures from your Procedure SuperBlocks as described in this chapter and in Chapter 3, *Ada Language Reference*. To link generated reusable procedures with your own application or simulator you have the three following options:

- Invoke the generated algorithmic procedure directly (refer to callout 2 of Figure 2-4), passing it pointers to structure objects as arguments.
- Invoke the re-entrant UCB wrapper routine generated to link with SystemBuild UCBs (refer to callout 3 of Figure 2-4), passing it arguments, which are pointers to arrays and values.
- Invoke the generated non-reentrant subsystem function, which invokes the procedure.

The first option is more efficient for performance reasons, but might not be the easiest to use. The second option provides ease of use in terms of argument list and re-entrancy, but is the least efficient in some cases. The third option also provides ease of use in terms of argument list, but although the procedure itself is re-entrant, the subsystem invoking the procedure is not re-entrant.

Invoking Generated Procedures Directly

To invoke a generated algorithmic procedure directly from your own application, you must have thorough understanding of the arguments to the procedure. When this is achieved, several steps need to be taken to invoke the procedure.

The model in Example 2-1 contains a Procedure SuperBlock named `proc` with a time delay block and a BlockScript block. The arguments to the generated procedure corresponding to Procedure SuperBlock `proc` are shown in Figure 2-5.

Example 2-1 Model with Procedure SuperBlock

Depending on the nature of the application, complete the following steps to invoke the procedure:

1. Create an object of type *`_procedure name_u`* and copy in the inputs to the procedure. A pointer to this object will be passed as argument U to the procedure.
2. Create an object of type *`_procedure name_y`* where the outputs of the procedure will be stored. A pointer to this object will be passed as argument Y to the procedure.

3. Create an object of type *_procedure name_*s where the states of the procedure will be stored and initialize all members of the object to 0.0. This should be done during initialization only. A pointer to this object will be passed as argument S to the procedure.

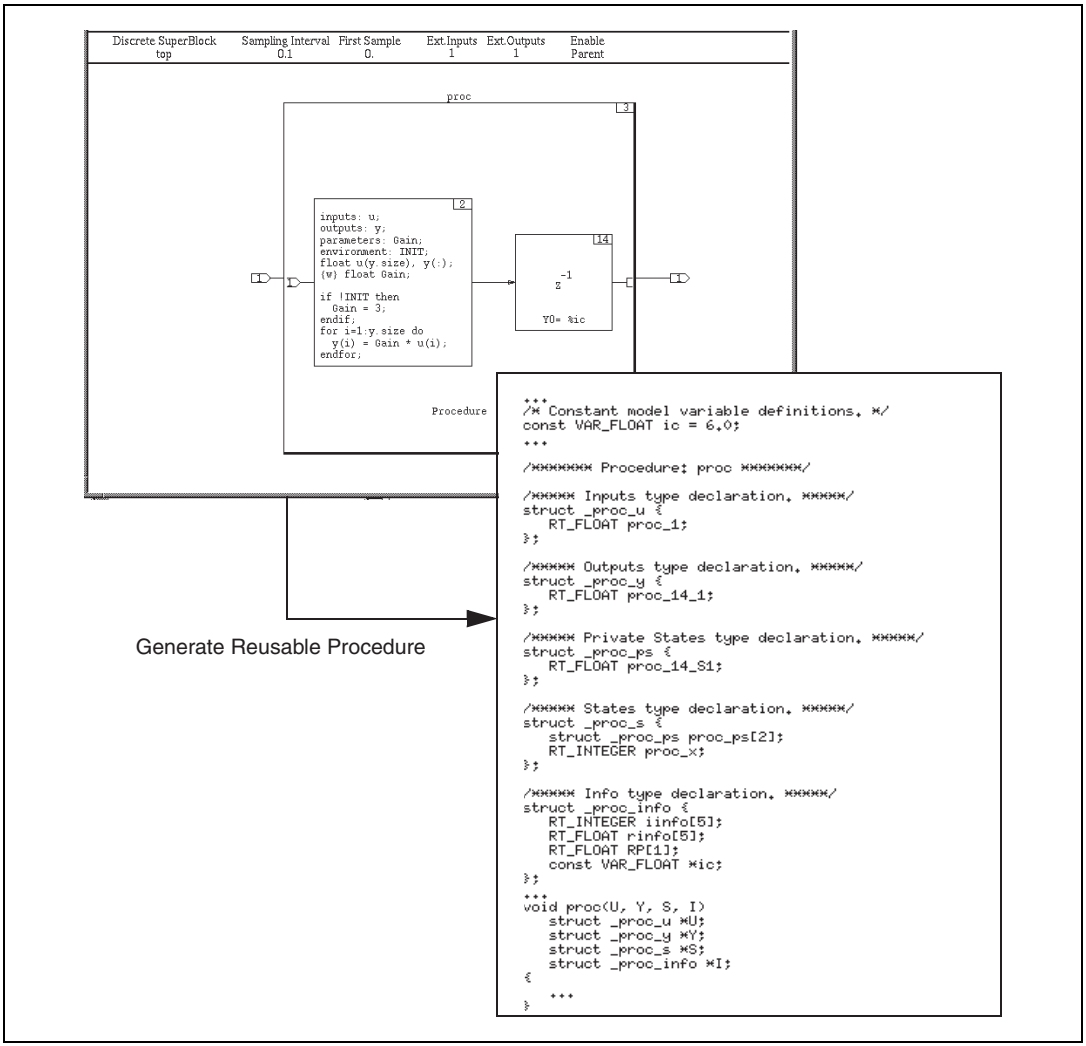


Figure 2-5. Arguments to Generated Procedure proc

4. Create an object of type *_procedure-name_*`info` where the informational data will be stored and assign proper values to elements `iinfo` and `rinfo`. If Xmath variables or Variable block variables were used in the procedure, the variable pointers need to be initialized to

point to the appropriate global variables. A pointer to this object will be passed as argument I to the procedure.

5. Invoke the procedure using pointers to the objects created in steps 1 through 4.
6. Toggle the state flag of the states object of the procedure—that is, if the value is 0 toggle it to 1, if the value is 1 toggle it to 0—before calling the procedure again.

Several of the previous steps are exercised by the other two methods for invoking generated procedures described in the *Linking Procedures with Real-Time Applications or Simulator* section. Details are provided in the *Invoking Procedures Using Generated UCB Wrapper Function* section and the *Invoking Procedures Using Generated Subsystem Function* section. Use the generated UCB wrapper and subsystem code as a guide. The UCB wrapper is generated for the purpose of re-entrancy, thus producing extra copy in and out of parameter and states variables and control and status arrays, while the subsystem code is not re-entrant—that is, the states and information data structures are declared as static such that the need to copy in and out of variables is no longer necessary.

Invoking Procedures Using Generated UCB Wrapper Function

As described in Chapter 1, *Introduction*, of the *AutoCode User Guide*, when generating a reusable procedure from a Procedure SuperBlock, a hook procedure or (UCB-style) wrapper along with the algorithmic procedure, is automatically generated. This wrapper is used by the SystemBuild simulator to automatically link the procedure for simulation. The arguments to the wrapper follow the same format of the SystemBuild explicit UserCode Block. A comment providing information about the wrapper and its arguments is generated above the wrapper function. You can use the wrapper directly from your application to invoke the procedure, but remember that this function is re-entrant, thus exercising copy in and out of variables that add extra overhead to the application.

Complete the following steps to invoke the wrapper function.

1. Create an array of four elements of type `integer`, representing the status and control argument `iinfo`, and initialize it properly. Refer to the *SystemBuild User Guide* for an explanation of `iinfo`. Only the first four elements of this array will be used by the generated procedure. This array will be passed as argument `iinfo`.
2. Create an array of four elements of type `double`, representing the timing-related information for the called procedure, and initialize it

properly. Refer to the *SystemBuild User Guide* for an explanation of `rinfo`. Only the first four elements of this array will be used by the generated procedure. This array will be passed as argument `rinfo`.

3. Create an array sized by the number of inputs in the procedure (refer to the comment) of type `double` and copy in the inputs to the procedure. This array will be passed as argument `U`. Also create a variable of type `int` and initialize to the number of inputs in the procedure. A pointer to this variable will be passed as argument `NU`.
4. Create an array sized by the number of outputs in the procedure (refer to the comment) of type `double` where the outputs of the procedure will be stored. This array will be passed as argument `Y`. Also create a variable of type `int` and initialize to the number of outputs in the procedure. A pointer to this variable will be passed as argument `NY`.
5. Create two arrays sized by the number of states in the procedure (refer to the comment) of type `double` and initialize all elements to 0.0. These arrays will be passed as arguments `X` (states) and `XD` (derivatives). Also create a variable of type `int` and initialize to the number of states in the procedure. A pointer to this variable will be passed as argument `NX`.
6. Create two arrays sized by the number of integer and real parameters in the procedure. Refer to the comment of types `int` and `double` and initialize all elements to 0 and 0.0, respectively. These arrays will be passed as arguments `I_P` and `R_P`.
7. Invoke the procedure using the arrays and pointers to the variables created in steps 1 through 6.

Invoking Procedures Using Generated Subsystem Function

When generating a reusable procedure from a Procedure SuperBlock, along with the algorithmic procedure and the (UCB-style) wrapper, a subsystem function (`subsys_`*`number`*) also is generated. You can use the subsystem function directly from your application to invoke the procedure, but keep in mind that this function is not re-entrant, as several variables in this function are declared static to avoid the overhead of copy in and out of the variables.

All of the following arguments need to be passed for each call to the procedure in the following order: `U`, `Y`. These arguments are pointers to structures reflecting the procedure's inputs and outputs. The inputs to the subsystem are provided by the argument `U`, a pointer to a structure named `_Subsys_`*`number`*`_in` (or `_Sys_ExtIn`). This structure has mixed

data-typed variables reflecting each subsystem input signal and type. The outputs to the subsystem are provided by the argument `Y`, a pointer to a structure named `_Subsys_`*`number`*`_out`. This structure has mixed data-typed variables reflecting each subsystem output signal and type.

The following overall steps need to be taken to invoke the subsystem function:

1. Create an object of type `_Subsys_1_in` (see generated subsystem code) and copy in the inputs to the subsystem. A pointer to this object will be passed as argument `U` to the subsystem.
2. Create an object of type `_Subsys_1_out` where the outputs of the subsystem will be stored. A pointer to this object will be passed as argument `Y` to the subsystem.
3. Invoke the procedure using pointers to the objects created in steps 1 and 2.

C Fixed-Point Arithmetic

Fixed-point calculations provide significant advantages over floating-point arithmetic. These include:

- Faster execution on most processors
- 8-bit, 16-bit, and 32-bit representations of fixed-point numbers
- Ability to interface to inexpensive processors that do not support floating-point arithmetic

This section describes the implementation of fixed-point arithmetic in AutoCode/C.

Note The *SystemBuild User Guide* has a fixed-point arithmetic chapter that explains the basics of fixed-point arithmetic and the use of fixed-point arithmetic in SystemBuild models.

Fixed-Point AutoCode/C Implementation

SystemBuild lets you represent vectors as fixed-point signals to which fixed-point arithmetic will be applied. Refer to the *SystemBuild User Guide*. Fixed-point signals and numbers are represented in AutoCode/C as integer data types. An associated radix position—that is, the integer marking the point that divides the integer and fractional part of the

number—for each integer item and the sign are managed by the code generator. Arithmetic expressions are scaled to emulate a fixed-point capability, and all expressions involving the item are coded to be consistent with the chosen radix position.

AutoCode supports a fixed-point library that implements all of the fixed-point operations (algebraic, relational, conversion). There are two different interfaces to the fixed-point library.

- The default and most commonly used one is the *macro interface* where the fixed-point operations are encoded as macros.
- The other interface is the *function interface* where all the fixed-point operations are implemented as functions. In this interface every fixed-point operation can result in a function call.

Note If you are using the function interface, compile the AutoCode-generated source file using the compiler flag `FX_FUNC_INTERFACE`.

For example, if your platform is Solaris and you are using the fixed-point function interface, the command line might appear as:

```
% acc -o gen_ap -DSOLARIS -DFX_FUNC_INTERFACE gen_ap.c
sa_*.o -lm
```

where *gen_ap.c* represents any AutoCode generated source file.

Because fixed-point operations get inlined while using the macro interface, an application linked with the macro interface will execute at a faster rate than the same application linked with the function interface.

All the files needed for the macro interface library are present in the directory `$CASE/ACC/macro_interface` while the files needed for the function interface are present in the directory `$CASE/ACC/function_interface`. By default, the system-specific source directory (`src`) is a link to the `macro_interface` directory. By linking and working in the directory `src`, the user gets to work with the macro interface. In order to work with the function interface, the user must make the `src` directory a link to the function interface. Files common to both directories are present in both directories.

The supplied macros and functions are written in C. C neither allows efficient coding of multiplication/division of 32-bit variables, nor does it take advantage of processor-specific capabilities, such as trapping of overflows, which can be detected in assembly code by checking the processor's status flag. However, you can write functions in assembly

language and replace the supplied macros (or functions) with your (assembly) functions so that you can take full advantage of the processor's arithmetic capabilities.

Generated Code with Fixed-Point Variables

Code generated for models using fixed-point variables—such as the examples provided in this chapter—will differ from code generated for models using floating-point, integer, or logical signals in the following areas:

- Signal and variable type declarations will reflect fixed-point types.
- Arithmetic operators `+`, `–`, `*`, and `/` will be replaced by fixed-point arithmetic macro calls (or function calls based on the interface used).
- Relational operators `>`, `>=`, `<`, `<=`, `==`, and `!=` will, when necessary, be replaced by fixed-point relational macro calls.
- Floating-point literals used in SystemBuild models will be replaced by the scaled integer counterpart.
- Macros (or procedures) for converting between various fixed-point types will be invoked when necessary.

Fixed-Point Data Types

Fixed-point type definitions are provided in the system-specific files `src` directory. Files `sa_types.h`, `sa_defn.h`, `sa_fxscale.h`, and `sa_fxlimit.h` use `typedef` statements to represent fixed-point types and related constants for 8-bit, 16-bit, and 32-bit data. All fixed-point types have an associated radix position value and a sign (signed or unsigned). The radix position value is clearly related to the data type scale factor $scale\ factor = 2^{-(radix\ position)}$. To perform any arithmetic operation, both the value and the radix position scalar are required. The table below lists the data types generated by AutoCode/C. For information on ranges and accuracy of each type, refer to the *SystemBuild User Guide*.

Table 2-6. AutoCode/C Data Types

Data Type	Number of Bits	Signed or Unsigned	Data Type Name
byte	8	unsigned	RT_UBYTE (radix 00) RT_UBYTExx (xx = radix 48 to –16)
		signed	RT_SBYTE (radix 00) RT_SBYTExx (xx = radix 48 to –16)

Table 2-6. AutoCode/C Data Types (Continued)

Data Type	Number of Bits	Signed or Unsigned	Data Type Name
short	16	unsigned	RT_USHORT (radix 00) RT_USHORTxx (xx = radix 48 to –16)
		signed	RT_SSHORT (radix 00) RT_SSHORTxx (xx = radix 48 to –16)
long	32	unsigned	RT_ULONG (radix 00) RT_ULONGxx (xx = radix 48 to –16)
		signed	RT_SLONG (radix 00) RT_SLONGxx (xx = radix 48 to –16)

A typical fixed-point type looks like the following:

RT_USHORT06

where USHORT stands for unsigned short, and 06 indicates the radix position.

Fixed-point variables that are always positive in nature can be declared as unsigned. This has the advantage of providing one more bit of accuracy than with signed fixed-point variables, because the most significant bit is used for storing the sign in that data type.

Example 2-2 shows some of the I/O type declarations. Only the significant parts of the code are shown.

Example 2-2 Fixed-Point C I/O Type Declarations

```
struct _Subsys_1_out {
 RT_SSHORT13 SS13;
 RT_SSHORT15 SS15;
 RT_SSHORT15 SS15_1;
 RT_USHORT14 US14;
 RT_USHORT10 US10;
 RT_SSHORT13 SS13_1;
 RT_SSHORT12 SS12;
 RT_SSHORT08 SS8;
```

```
 RT_SSHORT05 SS5;
 RT_SSHORT SS0;
 RT_SSHORT05 SS5_1;
};

struct _Sys_ExtIn {
 RT_USHORT13 US13;
 RT_SSHORT14 SS14;
};

/******* System Ext I/O type definitions. *******/
struct _Subsys_1_out subsys_1_out;
struct _Sys_ExtIn sys_extin;
static RT_FLOAT ExtIn [NUMIN+1];
static RT_FLOAT ExtOut [NUMOUT+1];

/******** Procedures' declarations ********/

/******* Procedure: proc *******/

/***** Inputs type declaration. *****/
struct _proc_u {
 RT_USHORT13 US13;
};

/***** Outputs type declaration. *****/
struct _proc_y {
 RT_SSHORT13 SS13;
 RT_SSHORT15 SS15;
 RT_SSHORT15 SS15_1;
 RT_USHORT14 US14;
 RT_USHORT10 US10;
 RT_SSHORT13 SS13_1;
 RT_SSHORT12 SS12;
};

/***** Info type declaration. *****/
struct _proc_info {
 RT_INTEGER iinfo[5];
};
```

User Types

You can define your own data types (called user types or UTs) in terms of the intrinsic types using the Xmath UT editor. UTs and the UT editor are described in the *SystemBuild User Guide*.

The UTs appear as `typedef` statements in the generated C code. For example:

```
typedef volts RT_SBYTE03;
```

This code defines the data type called volts to be a signed byte with radix position 3.

Overflow Protection

Overflow is defined as loss of significance—that is, a computation losing bits in the integer part of the number. The term *underflow* is used to mean overflow on a negative number.

An Overflow Protection capability is provided to protect against overflows. Overflow Protection consists in detecting an overflow condition and replacing the overflowed number with the extremal value for the data type of the number—the maximum positive number if the overflowed number was positive; the maximum negative number if it was negative.

Overflow can be efficiently detected in assembly code by examining the processor status flags, whereas in C, these flags are not available, making it necessary to test the results for consistency.

Most macros and functions with overflow protection have been combined into sets of signed and unsigned macros and functions, and combinations of both. This was done because overflow protection is different for signed and unsigned operands. This difference is due to the difference in lower and upper limits of signed and unsigned types. An unsigned type has 0 as the lower limit, whereas a signed type has a negative number as the lower limit. However, given the same word length and value position, the upper limit of a signed type is always smaller than the upper limit of an unsigned type.

Overflow protection is performed in all macros and functions that have a “p” at the end of the macro name. Examples of these macros are listed in the *Conversion Macros* section and the *Arithmetic Macros* section.

Overflow protection is controlled by the `-ovfp` option. Specifying `-ovfp 0` causes generation of unprotected macros, and specifying `-ovfp 1` causes generation of protected macros. The default is to generate protected macros.

Stand-Alone Files

All of the stand-alone files, with the exception of `sa_types.h`, have a common prefix `sa_fx`, indicating fixed-point functionality.

Macro Interface

The macro interface files are:

`sa_types.h`	Updated to include fixed-point types.
`sa_fx.h`	Contains fixed-point conversion macros.
`sa_fxp.h`	Contains fixed-point conversion macros with overflow protection.
`sa_fxr.h`	Contains fixed-point relational macros.
`sa_fxm.h`	Contains fixed-point arithmetic macros.
`sa_fxmp.h`	Contains fixed-point arithmetic macros with overflow protection. For 32-bit division and multiplication, overflow sometimes cannot be detected.
`sa_fx_temps.h`	Contains the declaration information for temporary variables used in fixed-point computations.
`sa_fxprv.h`	Contains macros used only by the other macros.
`sa_fxscale.h`	Contains scale factor constants for different radix values.
`sa_fxlimit.h`	Contains maximum and minimum values that can be represented in different fixed-point types.
`sa_fxadd_byte.c`	Contains fixed-point addition functions for byte data type.
`sa_fxadd_short.c`	Contains fixed-point addition functions for short data type.
`sa_fxadd_long.c`	Contains fixed-point addition functions for long data type.
`sa_fxsub_byte.c`	Contains fixed-point subtraction functions for byte data type.
`sa_fxsub_short.c`	Contains fixed-point subtraction functions for short data type.
`sa_fxsub_long.c`	Contains fixed-point subtraction functions for long data type.
`sa_fxmul_long.c`	Contains fixed-point multiplication functions for long data type. Multiplication for byte and short data types are completely implemented as macros.
`sa_fxdiv_long.c`	Contains fixed-point division functions for long data type.

`sa_fx_externs.c`	Contains definitions for extern variables such as mask buffers that are read only.

Function Interface

The function interface files are:

`sa_types.h`	Updated to include fixed-point types.
`sa_fxp.h`	Contains fixed-point conversion macros with overflow protection.
`sa_fxr.h`	Contains fixed-point relational macros.
`sa_fxm.h`	Contains fixed-point arithmetic macros.
`sa_fxmp.h`	Contains fixed-point arithmetic macros with overflow protection. For 32-bit division and multiplication, overflow sometimes cannot be detected.
`sa_fx_temps.h`	Contains the declaration information for temporary variables used in fixed-point computations.
`sa_fxprv.h`	Contains macros used only by the other macros.
`sa_fxscale.h`	Contains scale factor constants for different radix values.
`sa_fxlimit.h`	Contains maximum and minimum values that can be represented in different fixed-point types.
`sa_fx_f.h`	Contains prototypes for fixed-point conversion functions without overflow protection.
`sa_fxp_f.h`	Contains prototypes for fixed-point conversion functions with overflow protection.
`sa_fxm_f.h`	Contains prototypes for fixed-point algebraic functions without overflow protection.
`sa_fxmp_f.h`	Contains prototypes for fixed-point algebraic functions with overflow protection
`sa_fxadd_byte.c`	Contains fixed-point addition functions for byte data type.
`sa_fxadd_short.c`	Contains fixed-point addition functions for short data type.
`sa_fxadd_long.c`	Contains fixed-point addition functions for long data type.

`sa_fxsub_byte.c`	Contains fixed-point subtraction functions for byte data type.
`sa_fxsub_short.c`	Contains fixed-point subtraction functions for short data type.
`sa_fxsub_long.c`	Contains fixed-point subtraction functions for long data type.
`sa_fxmul_byte.c`	Contains fixed-point multiplication functions for byte data type.
`sa_fxmul_short.c`	Contains fixed-point multiplication functions for short data type.
`sa_fxmul_long.c`	Contains fixed-point multiplication functions for long data type.
`sa_fxdiv_byte.c`	Contains fixed-point division functions for byte data type.
`sa_fxdiv_short.c`	Contains fixed-point division functions for short data type.
`sa_fx_f.c`	Contains fixed-point conversion functions with overflow protection.
`sa_fxp_f.c`	Contains fixed-point conversion functions without overflow protection.
`sa_fxm_f.c`	Contains fixed-point algebraic functions without overflow protection.
`sa_fxdiv_long.c`	Contains fixed-point division functions for long data type.
`sa_fx_externs.c`	Contains definitions for extern variables such as mask buffers that are read only.

These stand-alone files have naming conventions based on the functionality of the macros and whether or not the macros support overflow protection. Refer to the *Overflow Protection* section. For example, the `sa_fxm.h` file contains macros performing arithmetic functions that do not have overflow protection, but file `sa_fxmp.h` contains macros with overflow protection that perform similar functions. There only is one file (`sa_fxr.h`) for relational macros as overflow protection is not a concern for macros implementing relational operations.

Fixed-Point Conversion and Arithmetic Macros

Although this section explains different fixed-point operations in terms of macros, all of these operations are supported as functions in the function interface. Hence, in the following sections the term *macro* can be completely substituted by the term *function*.

Three types of fixed-point macros are generated by AutoCode:

- Conversion macros that convert one type of number to another. Refer to the *Conversion Macros* section.
- Arithmetic macros that perform addition, subtraction, multiplication, or division. Refer to the *Arithmetic Macros* section.
- Relational macros that compare two numbers and produce a Boolean result. Refer to the *Fixed-Point Relational Macros* section.

Conversion Macros

Conversion macros are used for converting from one fixed-point data type to another, from floating-point to fixed-point or from fixed-point to floating-point, or from integer to fixed-point or from fixed-point to integer. These macros in turn make use of the left-shift or right-shift macros defined in `sa_fxprv.h`.

The right-shift macro shifts the bits of a value to the right. When a negative number is shifted right, it results in flooring of that number instead of truncating because of the two's complement scheme of representing negative numbers. Therefore, the right-shift macro with truncate behavior checks for a negative number and does the needed adjustment to produce truncation.

Whenever a fixed-point number needs to be aligned, a right-shift macro can be called. Addition, subtraction, multiplication, division, and the relational macros all make use of the alignment macros. Therefore, the right-shift macro can be heavily used. If you can accept floor behavior for negative numbers, you could replace the truncate macro with the floor macro, which can increase the execution speed of the generated code. To do so, modify the implementation of the (shift-right) macros `SHRsbpt`, `SHRsspt`, and `SHRslpt` in the `sa_fxprv.h` file to perform a simple right-shift operation as follows:

```
#define SHRsbpt(x,y) ((x) >> (y))
#define SHRsspt(x,y) ((x) >> (y))
#define SHRslpt(x,y) ((x) >> (y))
```

Figures 2-6 through 2-8 show how the conversion macros are named. Notice that macro names have no embedded spaces.

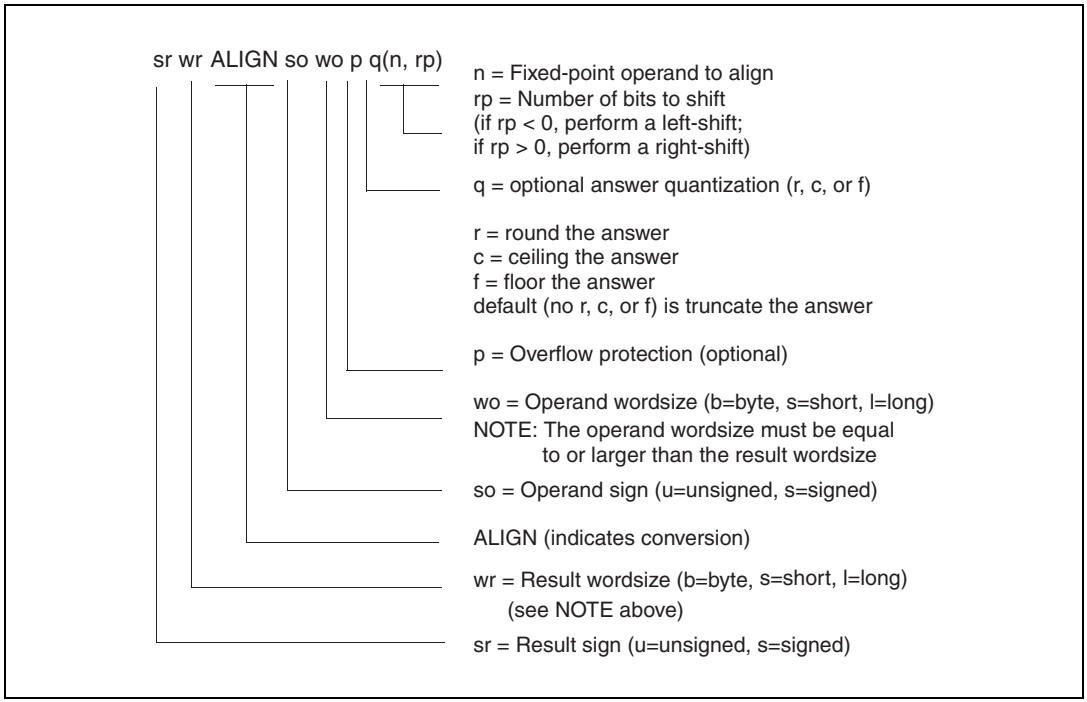


Figure 2-6. AutoCode/C Conversion Macros for Fixed-to-Fixed Conversions

For example, the macro to convert an unsigned 8-bit number to a signed 8-bit number with a shift of `rp` bits, and with overflow protection is:

```
sbALIGNubp(n,rp)
```

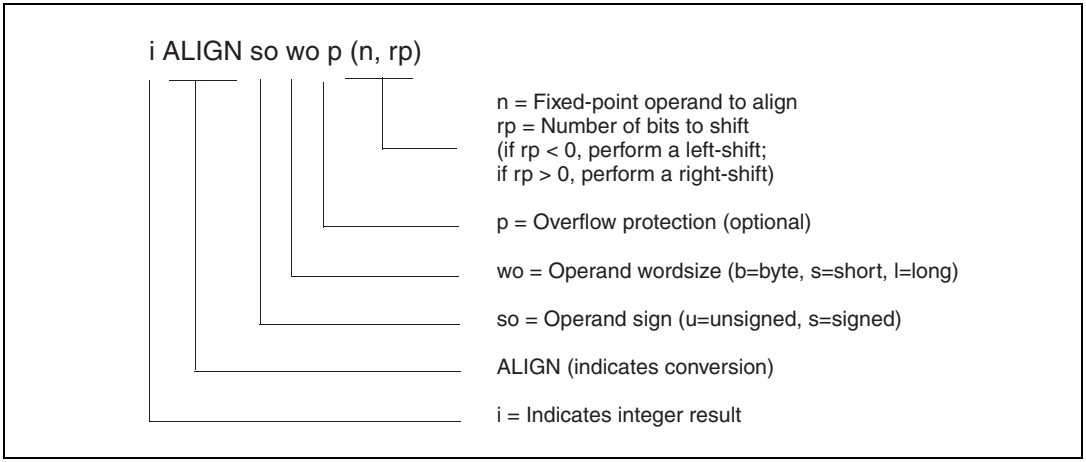


Figure 2-7. AutoCode/C Conversion Macros for Fixed-to-Integer Conversions

For example, the macro to convert an unsigned 8-bit number to an integer number with a shift of `rp` bits, and with overflow protection is:

```
iALIGNubp(n,rp)
```

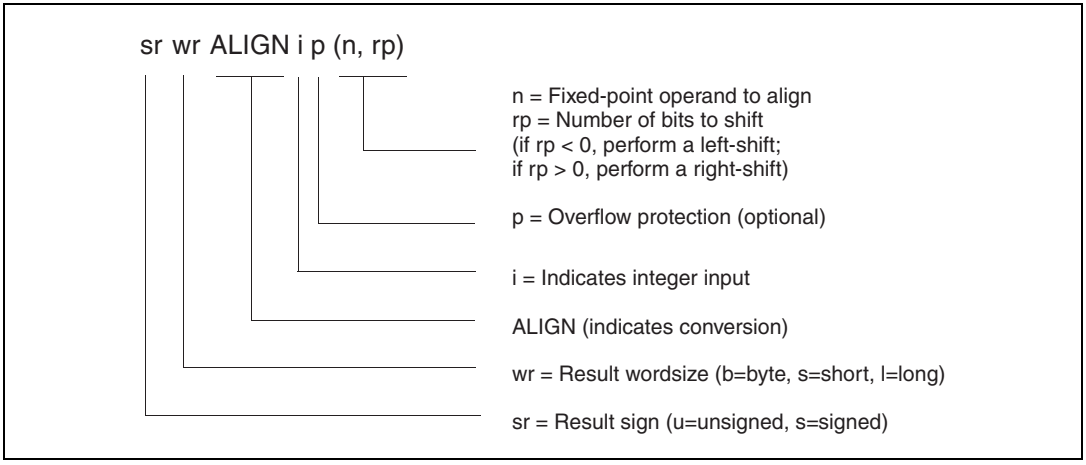


Figure 2-8. AutoCode/C Conversion Macros for Integer-to-Fixed Conversions

For example, the macro to convert an integer number to a signed 8-bit number with a shift of `rp` bits, and with overflow protection is:

```
sbALIGNip(n,rp)
```

Arithmetic Macros

The arithmetic macros perform addition, subtraction, multiplication, and division. The top level macros for arithmetic operations are present in the `sa_fxm.h` and `sa_fxmp.h` files. These macros in turn call the ALIGN macros that are defined either in `sa_fx.h` or `sa_fxp.h`, depending on whether or not they are overflow protected. The macros for addition and subtraction also make use of addition and subtraction functions defined in `sa_fxmp.c`.

Figure 2-9 shows how the arithmetic macros are named. Notice that macro names have no embedded spaces.

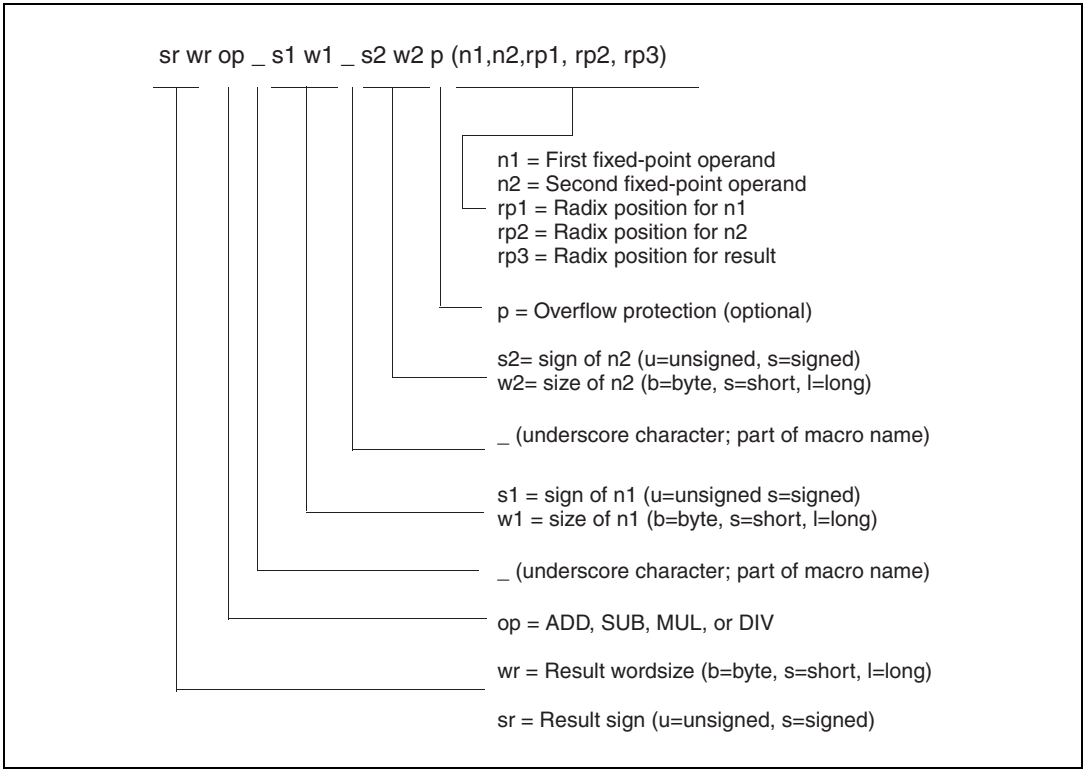


Figure 2-9. AutoCode/C Arithmetic Macros

Table 2-7 shows permissible operand and result sizes for the arithmetic macros.

Table 2-7. Arithmetic Macros—Operand and Result Sizes

Operation	Operand 1 Size	Operand 2 Size	Result Size
addition subtraction multiplication	byte	byte	byte or short
	short	short	short or long
	long	long	long
division (operand 1 is the dividend; operand 2 is the divisor)	byte	byte	byte
	short	byte	byte
	short	short	short
	long	short	short
	long	long	long

For example, the macro to add two 8-bit unsigned numbers with overflow protection and produce an unsigned 8-bit result is:

```
ubADD_ub_ubp(n1,n2,rp1,rp2,rp3)
```

The macro to subtract a 16-bit unsigned number from a 16-bit signed number with overflow protection and produce an unsigned 16-bit result is:

```
usSUB_ss_usp(n1,n2,rp1,rp2,rp3)
```

The macro to multiply two 16-bit signed numbers with overflow protection and produce a 16-bit signed result is:

```
ssMUL_ss_ssp(n1,n2,rp1,rp2,rp3)
```

The macro to divide two 32-bit signed numbers with overflow protection and produce a 32-bit signed result is:

```
slDIV_sl_slp(n1,n2,rp1,rp2,rp3)
```

Implementation of the Addition and Subtraction Macros

AutoCode has two implementations of the addition and subtraction macros:

- Macros that apply wordsize extension (also called extended intermediate types) to the two operands before aligning the radix positions and adding or subtracting. This is the default implementation. Wordsize extension provides greater accuracy, but is slower because the operations are performed in a larger wordsize than specified.
- Macros that do not apply wordsize extension.

For example, when using wordsize extension in an 8-bit processor, addition of two 8-bit operands results in a 16-bit addition operation, because the 8-bit addition macro internally extends the wordsize of the operands from 8 bits to 16 bits.

Not using the wordsize extension on 8-bit and 16-bit processors provides faster operations. However, you can lose precision if the result radix position is not smaller than the radix positions of the two operands. Alignment of the radix positions of the operands to the result radix position can overflow the 8-bit variable capacity, causing a saturation and loss of accuracy. Example 2-3 shows this.

Example 2-3 Using Wordsize Extension

Subtraction of fixed-point number `n1 = (17, r4)` and fixed-point number `n2 = (32, r5)`, to produce a fixed-point result number `n3` with radix position 7, using 8-bit signed variables for both the operands and the result.

Method 1: Using Wordsize Extension

In binary representation:

`0001^0001` (`n1 = (17,r4)`, decimal value = 1.0625)
–
`001^00000` (`n2 = (32,r5)`, decimal value = 1.0)

Extend the wordsize of `n1` and `n2` to 16 bits, and place the extended results in `n1'` and `n2'`:

`00000000 0001^0001` (`n1' = (17,r4)`, decimal value = 1.0625)
–
`00000000 001^00000` (`n2' = (32,r5)`, decimal value = 1.0)

Align the radix positions of n1 and n2 to the radix position of the result before subtracting (that is, shift n1 left by three bits, and shift n2 left by two bits). Place the aligned results in n1' and n2' and perform a 16-bit subtraction of n1' and n2', and store the result in n3':

```
00000000 1^0001000 (n1' = (136,r7), decimal value = 1.0625)
–
00000000 1^0000000 (n2' = (128,r7), decimal value = 1.0)
_____________________
00000000 0^0001000 (n3' = (8,r7), decimal value = .0625)
```

Change the result back to an 8-bit signed number in n3:

```
0^0001000 (n2 = (8,r7), decimal value = .0625)
```

Method 2: Not Using Wordsize Extension

In binary representation:

```
0001^0001 (n1 = (17,r4), decimal value = 1.0625)
–
001^00000 (n2 = (32,r5), decimal value = 1.0)
```

Align the radix positions of n1 and n2 to the radix position of the result before subtracting (that is, shift n1 left by three bits, and shift n2 left by two bits). Place the aligned results in n1' and n2' and perform an 8-bit subtraction of n1' and n2', and store the result in n3:

```
sign bit
|
0001^0001 (n1= (17,r4), decimal value = 1.0625)
```

Detect overflow before aligning to radix position 7, correct overflow (that is, use maximum number possible)

```
0^1111111 (n1' = (127,r7), decimal value = .9921875)
```

```
001^00000 (n2= (32,r5), decimal value = 1.0)
```

Detect overflow before aligning to radix position 7, correct overflow (that is, use maximum number possible)

```
0^1111111 (n2' = (127,r7), decimal value = .9921875)
```

```
0^1111111 (n1' = (127,r7), decimal value = .9921875)
–
```

`0^1111111 (n2' = (127,r7)`, decimal value = .9921875)

`0^0000000 (n3 = (0,r7)`, decimal value = 0.0)

In Example 2-3, method 1 is more accurate than method 2, but it is also less efficient because it involves a 16-bit subtraction. This is important for those using 8-bit processors but will probably not be as significant for those using 16-bit or 32-bit processors.

Method 2 was inaccurate because of the left-shifting that had to be performed for alignment to the result radix. If the result radix position had been the same as the radix position of one of the operands, the resultant value would have been as accurate as with method 1 even though it used only 8-bit subtraction.

Selecting Wordsize Extension in the Preprocessor Macro

You can choose whether or not to use wordsize extension in addition and subtraction by a preprocessor macro in the `sa_fxp.h` file. The preprocessor statement:

```
#define WORDSIZE_EXTEND 1
```

causes the code to be compiled with wordsize extension. This is the default. The preprocessor statement:

```
#define WORDSIZE_EXTEND 0
```

causes the code to be compiled without wordsize extension.

32-Bit Multiplication and Division Macros

32-bit multiplication and division macros are different from their 8-bit and 16-bit counterparts. This is because the maximum wordsize available is only 32 bits; therefore, operands cannot be promoted to a higher word length before or after performing multiplication or division.

32-Bit Multiplication

Before performing the actual multiplication, both operands are split into upper and lower words. Multiplication is performed in parts—that is, higher and lower words of each operand are multiplied by each other and added in a certain fashion to produce a 64-bit intermediate result. This intermediate result is aligned according to the result's radix position. After alignment, if the value cannot be held in 32 bits, the clipped value—that is,

the maximum possible value representable in 32 bits—is returned. This multiplication process can be expensive because it involves several multiplication and addition operations to produce an intermediate result. This procedure strives for accuracy, but a user can speed up the process by giving up some of the accuracy.

32-Bit Division

As with 32-bit multiplication, operands are split into higher and lower words. This method is based on Euclidean division or repeated division. The higher and lower words of the numerator are divided by the higher and lower words of the denominator. The remainder obtained from this step is repeatedly divided to get components of the quotient. These components are added up to get the final quotient. As with 32-bit multiplication, this can be costly because of several addition, division, and multiplication operations needed to calculate the intermediate results. Again, you can speed up the routine at the cost of accuracy.

16-Bit by 8-Bit Division

Depending on the radix value of the operands and the result, this operation might result in either an iterative division or a fast-shift-based division.

For example, let `n1` be the dividend with radix value `r1`, `n2` be the divisor with radix value `n2`, and `n3` be the result with radix value `r3`. If the term:

```
r1 - (r2) - (r3) ≤ BYTE_SIZE
```

where `BYTE_SIZE` is 8, it will result in a call to an iterative division. Otherwise, it will be a fast-shift-based division. The iterative division method is costly in terms of speed, but is needed to compute an accurate result. By changing this behavior, you can speed up the operation if you are willing to give up some accuracy.

32-Bit by 16-Bit Division

Depending on the radix value of the operands and the result, this operation might result in either an iterative division or a fast-shift-based division.

For example, let `n1` be the dividend with radix value `r1`, `n2` be the divisor with radix value `n2`, and `n3` be the result with radix value `r3`. The following term:

```
r1 - (r2) - (r3) ≤ WORD_SIZE
```

where `WORD_SIZE` is 16, results in a call to an iterative division. Otherwise, it will be a fast-shift-based division. The iterative division method is costly

in terms of speed, but is needed to compute an accurate result. By changing this behavior, you can speed up the operation if you are willing to give up some accuracy.

Note For more information on the implementation of multiplication and division macros, refer to the *SystemBuild User Guide*.

Fixed-Point Relational Macros

The relational macros compare two numbers and return a Boolean result (true or false). Macros for relational operations are defined in `sa_fxr.h`. These macros are defined in terms of low-level relational macros present in `sa_fxprv.h`.

Figure 2-10 shows how the relational macros are named. Notice that macro names have no embedded spaces.

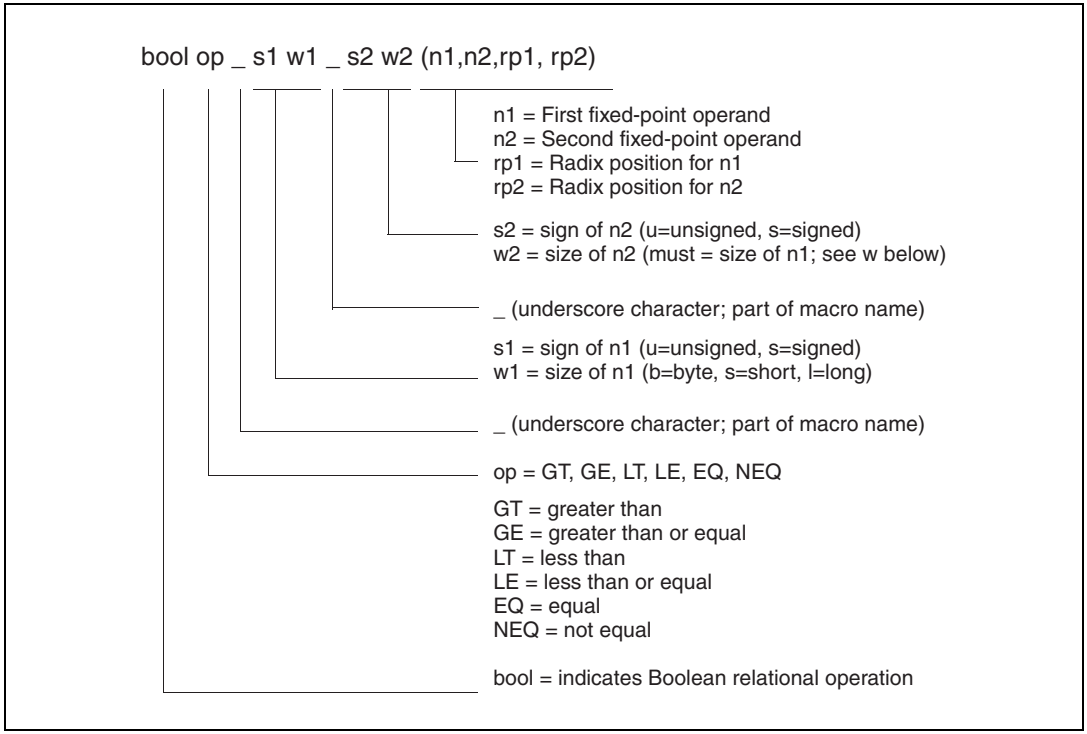


Figure 2-10. AutoCode/C Relational Macros

For example, the macro to check an 8-bit unsigned number and an 8-bit signed number for equality and produce a Boolean result is:

```
boolEQ_ub_sb(n1,n2,rp1,rp2)
```

Some Relevant Issues

- The fixed-point macros used by AutoCode-generated files are defined in the `sa` files and are available to you for making modifications. If the AutoCode macros are changed, the results might not match the Sim results. To change Sim so that the results again match, generate procedures-only fixed-point code (which uses the AutoCode fixed-point macros) and, through a UserCode Block (UCB), automatically link the generated code with the simulation engine. Refer to the *UserCode Block* section of Chapter 5, *Generated Code Architecture*.
- The fixed-point algebraic operations that involve more than two operands pose the problem of order dependency of the operation—that is, the operation can become nonassociative). For example, the expression $y = a + b + c$ can result in different values if evaluated as $(a + b) + c$ instead of $a + (b + c)$. Sorting the expression in a separate loop—for example, in the generated code for the model summing junction—is not performed by AutoCode due to the computational overhead. Refer to the *SystemBuild User Guide* for more details.
- Because the C preprocessor has a limit on the length of macros, it can process, the amount of nesting in the macros must be limited. Therefore, if a summing junction has more than six additions or subtractions, then in the generated code the additions or subtractions are broken down into multiple statements, and the intermediate result is stored in the destination signal (operand). For fixed-point computations, certain rules are used for coming up with the intermediate data type. If the summing junction is broken down into multiple statements in the generated code, the intermediate rules are influenced by the type of the destination signal (operand) present in each statement. This might result in some loss of accuracy.

3

Ada Language Reference

This chapter discusses files used to interface AutoCode and the generated Ada code to your specific platform and target processor. This chapter also discusses target-specific utilities needed for simulation and testing.

Stand-Alone Simulation

The templates provided for Ada code generation produce code that, when compiled, form a stand-alone simulation. This simulation can be executed with MATRIXx data as input, and produces results that can be loaded back into Xmath for analysis. You must compile the generated code along with the stand-alone library to produce the simulation executable.

Chapter 2, *Using AutoCode*, of the *AutoCode User Guide* describes a process to compile the code and stand-alone library, generate sample input data, and load the data into Xmath for analysis.

Supported Ada Compilers for the Stand-Alone Library

Ada'83 is extremely portable. The generated code from either of the standard templates is in complete conformance with Ada'83 and no vendor-specific extensions are used. However, there can be variations in the run-time environments supplied by Ada vendors that require different implementations, especially related to floating-point and fixed-point numerics. Three versions of Ada run-time environments that require slightly different implementations of the stand-alone library. If your Ada vendor is not specifically listed in Table 3-1, try one of the versions identified for your platform type. If the code compiles, it will most likely work. The names of the run-time environments correspond to the names of directories within the AutoCode distribution.

Table 3-1. Identified Ada Run-Time Versions of the Stand-Alone Library

Platform	Stand-Alone Library
Solaris, UNIX	Verdix
HP-UX/DOS	Alsys
Windows 2000/NT/9*x*/DOS	ActivAda

Supplied Templates

ada_rt.tpl Template

The `ada_rt.tpl` template is the default when generating Ada code. This template uses Ada tasks to implement the scheduling mechanism for the model. This template supports fixed-point arithmetic.

ada_sim.tpl Template

The `ada_sim.tpl` template does not use Ada tasks to implement a scheduler. The subsystems and procedures of a model execute as fast as possible. This template is similar in design to the `c_sim.tpl` for C code generation. The execution results of a model generated using this template should allow for easier comparison to SystemBuild Simulator results. Also, the time to execute a model is significantly faster than one based on the `ada_rt.tpl`.

ada_fxpt_sys.tpl Template

The `ada_fxpt_sys.tpl` template is included in the `ada_sim.tpl` and `ada_rt.tpl` templates. This template contains segments that generate the operator instantiation package. The segments have been modularized to reduce the impact of fixed-point support in the `ada_sim` and `ada_rt` templates. This template uses the system-level tokens to generate the operator instantiations.

ada_fxpt_sub.tpl Template

The `ada_fxpt_sub.tpl` template file can be included instead of the `ada_fxpt_sys.tpl` in the `ada_sim.tpl` and `ada_rt.tpl`. This template uses the subsystem/procedure-level tokens to generate the instantiations. Each subsystem and procedure has a different package and source file generated.

Stand-Alone Library

This section describes the system-specific and target-specific stand-alone (`sa_*`) files supplied with your system.

System-Specific Files

Header and source files are supplied to interface AutoCode generated code to your development platform, and to the target processor in your test bed or rapid-prototyping system.

Both specifications and the source files are provided in your source directory:

- Specification files have the suffix `_.a` or `_.ada` on most UNIX systems.
- Body files have the suffix `.a` or `.ada` on most UNIX systems.

The names of the distribution directories and files are shown in Table 3-2.

Table 3-2. Distribution Directories and Files

Category	UNIX	Windows
Top-Level Directory	`$MTXHOME` (Environment variable)	`%MTXHOME%` (Environment variable)
Executables	Directory: `$MTXHOME/bin` Executable: autostar	Directory: `%MTXHOME%\bin` Executable: autostar
Utilities	Directory: `$CASE/ACA/src` Files: `sa_*.a` or `sa_*.ada` Script: `compile_ada_sa.sh` Notes: A soft link to Alsys or Verdix directory. Directory: `$CASE/ACA/alsys` Files: `sa_*.ada` Notes: SA files for Alsys Ada compiler. Directory: `$CASE/ACA/verdix` Files: `sa_*.a` Notes: SA files for VERDIX Ada compiler.	Directory: `%CASE%\ACA\src` Files: `sa_*.a` Script: `compile_ada_sa.bat` Notes: A copy of the Verdix directory files. Directory: `%CASE%\ACA\alsys` Files: `sa_*.ada` Notes: SA files for Alsys Ada compiler. Directory: `%CASE%\ACA\verdix` Files: `sa_*.a` Notes: SA files for VERDIX Ada compiler.

Table 3-2. Distribution Directories and Files (Continued)

Category	UNIX	Windows
Templates	Directory: `$CASE/ACA/templates` Templates: `ada_rt.tpl`, `ada_intgr.tpl`, `ada_sim.tpl` Compiled Template: `ada_rt.dac`, `ada_sim.dac`	Directory: `%CASE%\ACA\templates` Templates: `ada_rt.tpl`, `ada_intgr.tpl, ada_sim.tpl` Compiled Template: `ada_rt.dac`, `ada_sim.dac`
Demos	Directory: `$XMATH/demos`	Directory: `%XMATH%\demos`

- The principal file is `sa_utils.a` (or `sa_utils.ada`), the stand-alone utilities file. When you compile `sa_utils.a/.ada`, you must make the following files from the source distribution directory available locally:
  - `sa_utils_.a/.adasa_utils.a/.ada`
  - `sa_time_.a/.adasa_time.a/.ada`
  - `sa_math_.a/.adasa_math.a/.ada`
  - `sa_fmath_.a/.adasa_fmath.a/.ada`
  - `sa_types_.a/.adasa_io.a/.ada`
  - `sa_defn_.a/.ada`
- If you are compiling to run with the Real-Time/Expert Block, the following files must be available locally:
  - `sa_exprt_.a/.adasa_exprt.a/.ada`
  - `sa_aiq_.a/.adasa_aiq.a/.ada`
- If you are compiling to run with the Real-Time/Fuzzy Logic Block, the following files must be available locally:
  - `sa_fuzzy_.a/.adasa_fuzzy.a/.ada`
- If you have defined any hand-written UserCode Blocks, these header files must be available locally:
  - `sa_user_.a/.adasa_user.a/.ada`
- When you compile your generated code program, all of the previous files must be available.

For use with generated Ada code, mathematical functions are supplied in the `sa_fmath.a/.ada` file, using type `RT_FLOAT`. A set of auxiliary math functions that are needed to support AutoCode/Ada generated programs can be found in the AutoCode/Ada supplied package `sa_math.a/.ada`. These functions are required even if they are supported by your native Ada

floating-point MATH_LIB. The sa_time.a/.ada file provides the Elapsed_Time_Of( ) function.

The purposes of the more important specification files are listed in Table 3-3.

Table 3-3. Target-Specific Utility Routines

File	Purpose
sa_types_.a	Defines the supported data types.
sa_defn_.a	Defines constants and error codes for generated code.
sa_utils_.a	Defines external function prototypes for the stand-alone utilities.
sa_math_.a	Defines special math functions used by generated code.
sa_fmath_.a	Used only by Verdix compilers to re-export functions already supported by the Ada math library.
sa_time_.a	Declares time-related variables and functions.
sa_user.a	Defines function prototypes for UCBs.

Data Types

Several of the target-specific utilities are involved with data types; three data types (declared in sa_types.a/ada) are defined for the Ada Code Generator:

RT_FLOAT	Corresponds to Ada type FLOAT.
RT_INTEGER	Corresponds to Ada type INTEGER.
RT_BOOLEAN	Corresponds to Ada type BOOLEAN.

At compilation, you must make available the specification file sa_types_.a (or sa_types_.ada), which declares these types, along with corresponding array types. This file is in the source distribution directory on your system and you can edit a copy of the file as required. Certain global record, array, and exception types are also defined in this file. The record type RT_STATUS_RECORD is declared in sa_types_.a/.ada and is used when UserCode Blocks are referenced.

Target-Specific Utilities

The target-specific utilities in the `sa_utils.a` or `sa_utils.ada` file perform hardware-, application-, and Ada-specific tasks that are required for simulation and testing. They can be modified to support the generated code on the target computer. As furnished, these routines simulate I/O operations that would occur in a simulation environment, using import files created using MATRIXx. These files are to be used unmodified for comparing your simulations with generated code outputs. However, for target-system usage on your rapid prototyping or end-user system, these routines can be modified or rewritten completely and recompiled for the target system. When you do this, be sure to keep a copy of the unmodified file and keep separate versions of the files in separate directories. There is no requirement that the file be named `sa_utils.a`; however, the name you use must be specified for compilation and linking. Inside the file, the names of all the external variables, functions, and other references must be preserved.

As furnished for this release, the routines are written in Ada, but this is not required. If you rewrite the routines, they should still be written in a language that offers direct control of the I/O and interrupt hardware of the target computer and that can be linked to the object form of the generated Ada program. Normally, these utilities need to be created only once for each target computer. In general, a given set of target-specific utilities need only be changed when the target hardware is modified. The routines are shown in Table 3-4.

Table 3-4. Target-Specific Utility Routines

Routine	Function
`Enable( )`	Unmask timer interrupt.
`Disable( )`	Mask timer interrupt.
`Background( )`	Background polling loop.
`Error,fatalerr( )`	Error handlers.
`Implementation_Initialize( )`	Initialize I/O hardware and perform other implementation-specific initialization tasks.
`Implementation_Terminate( )`	Perform implementation-specific termination tasks.
`External_Input( )`	Collect external inputs.

Table 3-4. Target-Specific Utility Routines (Continued)

Routine	Function
`External_Output( )`	Post external outputs.
`Signal_Remote_Dispatch( )`	Multiprocessor implementations only; signal secondary processors that a dispatch table is available.

The `sa_utils.a` or `sa_utils.ada` file contains comments about each of the routines as they are used for comparing simulation with generated code results.

Enable( ), Disable( ), and Background( ) Procedures

Enable, disable, and background routines are not needed when the application scheduler is written in terms of the Ada tasking model. They are furnished as null stubs in the `sa_utils.a` or `sa_utils.ada` file. If you have a system that requires intervention in this area, place the required code in these procedures.

Error Procedure( ) Procedure

```
procedure Error(NTSK: in RT_INTEGER;
ERR_NUM: in RT_INTEGER)
```

`Error( )` reports conditions detected by the generated code during execution; not all reported conditions are errors. These functions can be invoked for deactivating all necessary functions and then passing an alarm to the external environment or for initiating recovery action. You can choose either to return from an error function or to halt the machine.

The `RT_INTEGER` variable `ERR_NUM` is passed if an error occurs in running the generated code. The following conditions are trapped. Not all of them necessarily indicate that an error has occurred. The following messages may be generated during the execution of the generated code:

`Stop Block encountered in task n`

This is not necessarily an error. This refers to a SystemBuild Stop Simulation Block, encountered in the execution of the generated code.

`Math Error encountered in task n`

Check your model for possible overflows, divisions by zero, and so on.

`User code error encountered in task n`

Refer to the Chapter 14, *UserCode Blocks*, of the *SystemBuild User Guide* or the source listing of the USR01 routine for meanings of UCB errors. You are allowed to extend the scope of these messages, so it might be one of yours.

Unknown error encountered in task *n*

A possible cause is an incorrect user-written error condition in the generated code.

Time overflow occurred in task *n*

This indicates a subsystem (or task) overflow. Also, you might get this error if the generated real-time code is run on non-real-time systems.

Unexpected error in task *n*

This message occurs if an error other than any of the previous examples is encountered.

Implementation_Initialize( ) Procedure

```
procedure Implementation_Initialize
(NUMIN:in RT_INTEGER;
 NUMOUT:in RT_INTEGER,
 SCHEDULER_FREQ:in RT_FLOAT)
```

In the default (simulation comparison) version of the `sa_utils.a/.ada` file, this function initializes the inputs and outputs for the system by loading input data from the user-furnished `Xmath` {`matrixx`, `ascii`} formatted input file. In the version of this routine that you write to make the generated code work with the target computer, this routine performs many kinds of implementation-specific initialization processes for the real-time system. These processes include:

- Initializing the clock-timer of the target system to request interrupts at the minor cycle of the control system; that is, the time period required between interrupts for synchronous operation of the various tasks, as calculated by AutoCode from the block diagrams
- Initializing the interrupt system of the target computer
- Initializing the math coprocessor, if any
- Setting up shared memory and other hardware-dependent processes
- Initializing I/O hardware

By default, several error conditions are trapped in the procedure `Implementation_Initialize` of `sa_utils.a/.ada`, but you can expand this capability in your own versions of the program, detecting your own error conditions and adding your own messages.

The Ada exception processing facility is used and you are encouraged to define and raise exceptions in your versions of the `Implementation_Initialize` procedure.

The generated messages displayed in the default version of the `sa_utils.a/.ada` file are listed below. These messages pertain to the processing of the input and output files for execution of generated code:

***** File opened is not in Xmath {matrixx, ascii} format.**

Load the file into Xmath, save in {`matrixx`, `ascii`} format, then try again.

***** Incorrect file version. Must be at least V7.0.**

The input data file was generated using an obsolete version of MATRIXx. Load the file into Xmath, save it from the Xmath Commands window, and try again. Notice that `V7.0` refers to the version of the file save routine, not to the version of MATRIXx.

***** Invalid file name**

Check to see that the correct Ada file naming conventions are used.

***** Error opening input file**

Check to see that the correct Ada file usage conventions are used.

*****Input file contains more than two arrays.**

*****Input time vector has more than one column.**

*****Input time vector cannot be an imaginary number.**

*****Input array dimensions must be TIME_POINTS X NUMBER OF INPUTS**

*****Input array cannot contain imaginary numbers.**

All the previous messages indicate a bad input file.

***** First time point must be zero.**

An input time vector must start at zero.

***** Time vector size exceeds storage limit.**

***** Input array size exceeds storage limit.**

These messages indicate that the sizes of the time vector and input array have exceeded one or more of the storage allocation size limits established by `sa_utils.a/.ada`. These limits are established as constants at the very beginning of the stand-alone utilities file, just after the trademark notice. Refer to the comments in the file and adjust the limits accordingly; then recompile and relink the utilities file. If the input time vector size is adjusted, the output time vector must be adjusted to maintain its size as twice the input time vector. As a rule of thumb, the maximum input values should be equal to the maximum output values, plus the maximum time outputs.

`*** Output storage exceeds storage limit.`

The size of the input file has exceeded the storage allocation size limits established by `sa_utils.a/.ada`. This limit is established as a constant along with the input size limits discussed previously. Refer to the comments at the beginning of the file and adjust the limits accordingly, and then recompile and relink.

Implementation_Terminate( ) Procedure

```
procedure Implementation_Terminate
```

In the default simulation comparison version of the `sa_utils.a/.ada` file, this procedure completes the I/O processing for the execution of the system by writing output data to the output file and closing the file.

You can write a version of this routine to make the generated code work with the target computer. In addition to data completion tasks, this routine can be called upon to perform many kinds of implementation-specific shutdown processes for the real-time system. These include:

- Freeing up shared memory and other resources
- Completing the posting of all outputs
- De-initializing I/O hardware

External_Input ( ) Procedure

```
procedure External_Input(Bus: out RT_REAL_ARRAY;
UE_PTR: in RT_INTEGER;
NUMIN: in RT_INTEGER;
SCHEDULER_STATUS: out INTEGER)
```

`External_Input( )` is for use in simulation comparison mode. It samples input data values from an input pool previously loaded into memory. In the target version of `sa_utils.a/.ada`, the operation of

External_Input( ) is much the same; it returns an input vector from the software data bus. External_Input( ) returns the value of SCHEDULER_STATUS, which is passed to it by the scheduler.

External_Output( ) Procedure

```
procedure External_Output(Bus: in RT_REAL_ARRAY;
YE_PTR: in RT_INTEGER;
NUMOUT: in RT_INTEGER;
SCHEDULER_STATUS: out INTEGER)
```

External_Output is for use in simulation comparison mode. It places output data values into an output pool in memory. In the target version of sa_utils.a/.ada, the operation of External_Output( ) is much the same; it posts an output vector to the software data bus. External_Output( ) returns the value of SCHEDULER_STATUS, which is passed to it by the scheduler.

UserCode Blocks

This section describes how to link UserCode Blocks (UCBs) with AutoCode or SystemBuild applications.

Linking Handwritten UCBs with AutoCode Applications

To write the implementation code for UserCode Blocks (UCBs), refer to the sa_user_.a/.ada and sa_user.a/.ada files, which are provided in your src directory. These files contain an example of the declaration of the calling sequence described in the *Calling UCBs* section. Make one or more copies of these files and insert your code that implements the functionality of the UCBs. You can place one or more uniquely named UCB code procedures inside each copy of the sa_user_.a/.ada and sa_user.a/.ada files and give the copies any convenient names. If you include renamed files, place the names in the stand-alone file compilation script (compile_ada_sa.sh) for automatic compilation.

The computations performed by UCBs can update both block states and outputs.

Calling UCBs

Every one of the following arguments will be passed for each call to the UCB in the following order:

```
INFO, T, U, NU, X, XD, NX, Y, NY, R_P, I_P
```

Within the UCB, the elements of all the array variables (`U`, `X`, `XD`, `Y`, `R_P`, `I_P`) must be accessed as in the following example:

```
U(U'first), U(U'first+1), ... U(U'first+NU-1)
X(X'first), X(X'first+1), ... X(X'first+NX-1)
```

Make sure to access the elements in the above manner as all the arrays are passed as array slices. The sizes of `R_P` and `I_P` must be entered in the UCB block form to ensure that proper storage is allocated in the caller. Also, the initial conditions for states have to be specified in the form. Table 3-5 lists the type and purpose of UCB arguments used in the fixed calling method.

Table 3-5. UCB Calling Arguments and Data Types

Argument	Data Type	Description
`INFO:in out`	`RT_STATUS_RECORD`	A structure representing operation requests and status.
`T:in`	`RT_DURATION`	Elapsed time.
`U:in`	`RT_REAL_ARRAY`	An array (of size NU) of inputs to the UCB.
`NU:in`	`RT_INTEGER`	The number of inputs to the UCB.
`X:in out`	`RT_REAL_ARRAY`	An array (of size NX) of state variables of the UCB.
`XD:in out`	`RT_REAL_ARRAY`	An array (of size NX). Defines the next discrete states of X for discrete subsystems, and the derivative of X for the continuous subsystems.
`NX:in`	`RT_INTEGER`	The number of states (and next states).
`Y:in out`	`RT_REAL_ARRAY`	An array (of size NX). Defines the next discrete states of X for discrete subsystems, and the derivative of X for the continuous subsystems.
`NY:in`	`RT_REAL_ARRAY`	The number of outputs from the UCB.
`R_P:in out`	`RT_REAL_ARRAY`	An array of real parameters.
`NRP:in out`	`RT_INTEGER`	The number of real parameters.

Table 3-5. UCB Calling Arguments and Data Types (Continued)

Argument	Data Type	Description
I_P:in out	RT_INTEGER	An array of integer parameters.
NIP:in out	RT_INTEGER	The number of integer parameters.

The operations within UCBs are controlled by the INFO argument, a record of RT_STATUS_RECORD type that is passed as part of the argument list for each UCB:

```
type RT_STATUS_RECORD is
record
ERROR: RT_INTEGER;
INIT: RT_BOOLEAN;
STATES: RT_BOOLEAN;
OUTPUTS: RT_BOOLEAN;
end record;
```

The following example shows the general form of UCB equations and indicates how the INFO status record is used to control the computations.

```
if INFO.INIT then
  -- do user code initialization
   INFO.INIT :=  FALSE;
end if;
if INFO.OUTPUTS then
  -- do output calculations having the general form:
  -- Y := h(T,X,XD,U,R_P,I_P);
end if;
if INFO.STATES then
  -- do state update calculations with the general form:
  -- XD := f(T,X,XD,U,R_P,I_P);
end if;
```

When an error occurs within the UCB, set INFO.ERROR:= some nonzero integer value as an error return. Also, make sure that INFO.INIT is set to FALSE at the end of the initialization cycle.

To link UCBs (either handwritten or generated) with generated scheduled subsystem code, compile the UCBs, required sa_* files, and the generated scheduled subsystem code and link them together to build an application.

Procedure SuperBlocks

This section describes how to generate and link Procedure SuperBlocks.

Generating Reusable Procedures

Generate reusable procedures from your Procedure SuperBlocks as described in Chapter 2, *Using AutoCode*, of the *AutoCode User Guide*. You have an option to call the generated algorithmic procedures directly (refer to Example 3-1), passing it pointers to record objects as arguments, or you can call the hook routine with a *fixed* UCB calling style interface, passing it arguments, which are arrays and values. The former option is more efficient for performance reasons, but the latter provides for backward compatibility in terms of argument list.

Refer to Chapter 5, *Generated Code Architecture*, for more information.

Linking Procedures with Real-Time Applications or Simulator

Linking Ada procedures (either handwritten or generated) back with SystemBuild for simulation requires the use of pragmas or implementation-dependent features for calling Ada subprograms from languages other than Ada (refer to Example 3-1).

Example 3-1 Linking Generated Reusable Procedures

```
-------- Procedure: myproc --------
...
package myproc_pkg is
   ...
   procedure myproc(U : myproc_u_t_P;
                    Y : myproc_y_t_P;
                    I : myproc_info_t_P);
end myproc_pkg;
...
package body myproc_pkg is
   procedure myproc(U : myproc_u_t_P;
                    Y : myproc_y_t_P;
                    I : myproc_info_t_P) is
      ...
   begin
      -- Body of the procedure --
   end myproc;
end myproc_pkg;
```

```
------------------------------------------------------------------------
-- Wrapper around the procedure to support UCB (Fixed calltype) interface
-- This routine can be called as a UCB either from SystemBuild or AutoCode
------------------------------------------------------------------------
------------------------------------------------------------------------
-- Map
-- ---
-- Number of Inputs:                   2
-- Number of Outputs:                  3
-- Number of States:                   0
-- Number of Integer parameters:       0
-- Number of Real parameters:          0
-- ...
------------------------------------------------------------------------
...
with myproc_pkg;            use myproc_pkg;
procedure myproc_hook(
            iinfo     : in out RT_STATUS_RECORD;
            TIME      : in     RT_DURATION;
            U         : in     RT_FLOAT_AY;
            NU        : in     RT_INTEGER;
            X         : in     RT_FLOAT_AY;
            XD        : in out RT_FLOAT_AY;
            NX        : in     RT_INTEGER;
            Y         : in out RT_FLOAT_AY;
            NY        : in     RT_INTEGER;
            R_P       : in out RT_FLOAT_AY;
            I_P       : in out RT_INTEGER_AY) is
   ...
begin
   ...
   myproc(ptr_of(myproc_12_u'address), ptr_of(myproc_12_y'address),
          ptr_of(myproc_12_i'address));
   ...
end myproc_hook;
```

Ada Fixed-Point Arithmetic

This section describes the implementation of fixed-point arithmetic in AutoCode/Ada. It is assumed that you are familiar with the Ada programming language.

Note The *SystemBuild User Guide* has a fixed-point arithmetic chapter that explains the basics of fixed-point arithmetic and the use of fixed-point arithmetic in SystemBuild models.

How to Generate Real-Time Code

Using AutoCode, you can generate Ada high-level language code. Refer to the *Template Programming Language User Guide* or to Chapter 2, *Using AutoCode*, of the *AutoCode User Guide* for additional information. To generate code to run on your local host, you can generate code from one of the following:

- SystemBuild, which lets you automatically generate a real-time file (`.rtf`) and then source code from a model, using a Graphical User Interface. This is the recommended method of generating code to run on your local host.
- Xmath, which lets you automatically generate an `.rtf` file and then source code from a model, using an Xmath command.
- The operating system prompt, which lets you generate source code from an already-existing `.rtf` file, using the `autostar` command from the operating system prompt.

Fixed-Point AutoCode/Ada Architecture

This section describes the architecture of Fixed-Point AutoCode/Ada. Consult an Ada language reference manual if you are not familiar with any of the terms in this section. The basis for this architecture is the use of the fixed-point type mechanism provided in Ada. This basis enables the use of generic functions to implement the functionality of standard operations, such as addition and subtraction. Overloaded operators are created as instances of the generic functions for only those combinations of fixed-point data types used in the model. The use of overloaded operators maximizes code readability.

Fixed-Point Data Types

Fixed-point type declarations exist in the file named: `sa_fxptypes_.a` and is provided in the System-Specific Files `src` (source) directory.[1] This file contains the specification of the `SA_FIXED` package. The package specification contains all of the type declarations with the appropriate delta and range for each type. Refer to the *AutoCode Reference* for more information about the fixed-point type sizes, radix ranges and naming conventions.

Note Ada uses the term delta to describe the accuracy of a fixed-point type. AutoCode/Ada uses the term radix to denote a type's accuracy. The relationship between the two is: delta = 2^{-radix}. Thus, a fixed-point type with a delta of 0.125 is a fixed-point type with radix 3.

Generic Functions

Generic functions that implement functionality of standard operations are found in the two files named: `sa_fxpgenerics_.a` and `sa_fxpgenerics.a`. These files provide the specification and body for the `SA_FIXED_GENERICS` package. These files are also provided in the System-Specific Files `src` directory. These generic functions are the basis for the creation of the overloaded operators to perform the appropriate arithmetic operation.

Instantiated Functions

Due to the large number of combinations of operations and fixed-point types, only those operations that are used in the model are instantiated—that is, only the instances of the functions that are required by a model are actually created. Thus, an additional file is created by the code generator when generating code for a model that uses fixed-point types. This file is generated from the template. The default template generates one additional file that contains a package specification containing all of the instantiated functions for the model. The package name is `RT_FIXED_OPERATORS` and the file is named `fxp_`*`outputfile`*`_.a`, where *`outputfile`* is the name of the model or other name as specified in a command option to AutoCode.

[1] The file name convention uses an underscore before the extension to denote a file containing a package specification. The `.a` extension is arbitrary as different platforms use different extensions, like `.ada`.

Package Dependencies

The fixed-point AutoCode/Ada architecture forces a dependency among the supplied and generated code for a model that uses any fixed-point types. This dependency affects compilation order within an Ada Library. Figure 3-1 illustrates the imposed dependency. In the figure, a package that is beneath a package and connected to a package above it by an arrow is said to be dependent on the package that is above.

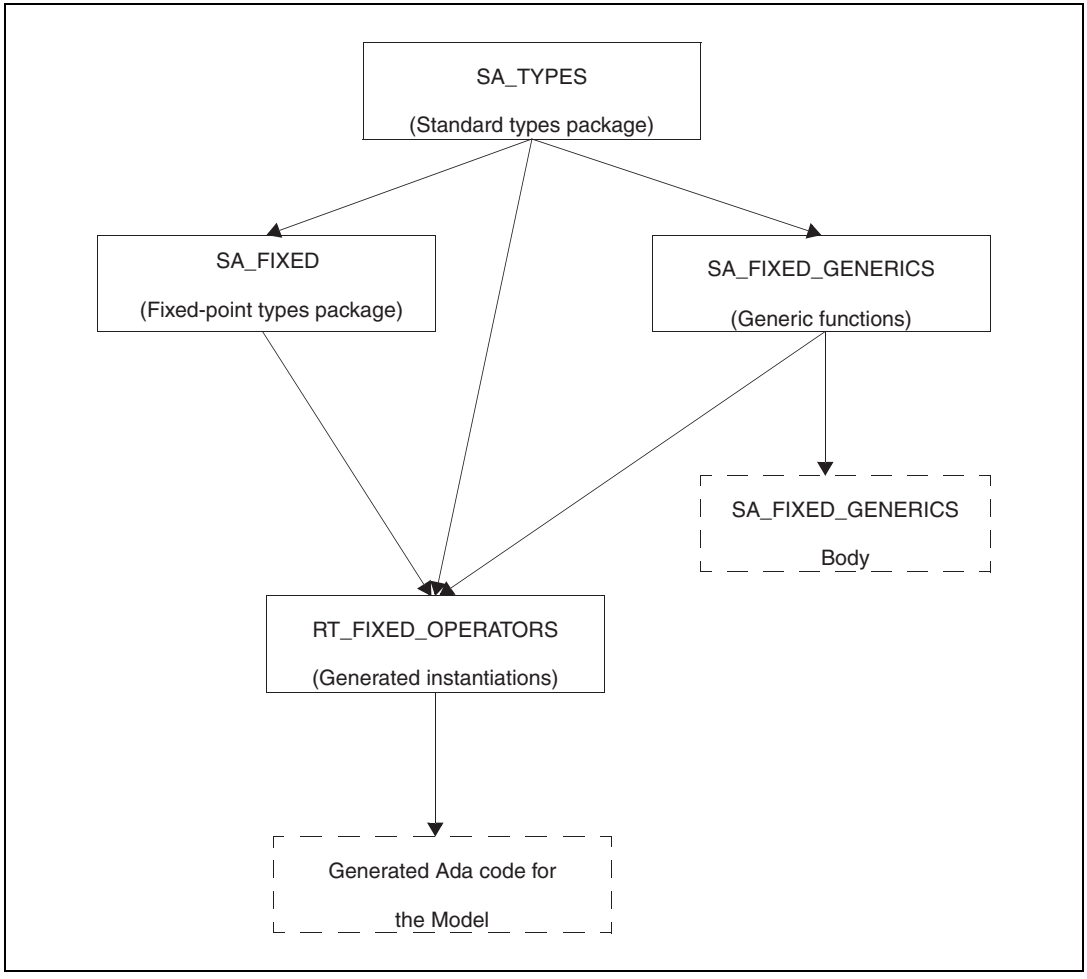


Figure 3-1. Package Dependency Graph

Generated Code with Fixed-Point Variables

Fixed-point arithmetic operations are accomplished by overloading the standard arithmetic operators for the fixed-point types. Therefore, the generated code with fixed-point variables uses the same infix expressions that non-fixed-point variables normally use, except for the following modifications:

- The package name must be used when referring to a fixed-point type.
- Explicitly defined conversion functions are used to convert a fixed-point value to any other numeric type.
- No-Op conversion-like functions are used to disambiguate infix subexpressions.
- Initialization of fixed-point variables with negative literals use the pre-defined negate operator to specify a negative value.
- Tests for equality with a fixed-point type are coded using equivalent logical expressions. The expression `a = b` will be generated as `(a >= b and a <= b)`, and the expression `a /= b` will be generated as `(a < b || a > b)`, when `a` and/or `b` is a fixed-point type variable or expression.

User Types

A User Type (UT) is a type that is specified using the Xmath User Type editor and appears in the generated code as a type name. UTs and the UT editor are described in more detail in the *SystemBuild User Guide*.

For generated Ada code, all UTs are placed in a package named `USER_TYPES`. The package declares a UT as a subtype of the base type of the UT. For example, the UT named volts is declared in the `USER_TYPES` package as shown in Figure 3-1.

```
subtype volts is SA_FIXED.RT_SBYTE03;
```

In other words, the data type `volts` is a subtype of the signed byte with radix 3 fixed-point type. Notice that UTs are not restricted to fixed-point base types. The package name must be used to refer to a user type for declaring variables.

System-Level Parameters to Generate User Types

Table 3-6 describes new system-level parameters that are used to generate the USER_TYPES package.

Table 3-6. System-Level Parameters to Generate User Types

Name	Description
n_user_defined_type_i	The number of user types in the current model.
usertype_name_ls	An array of strings that contain the names of all of the user types in the model. Array size is n_user_defined_type_i.
usertype_basename_ls	An array of strings that contain the names of all the base types of the user types of the model. Array size is n_user_defined_type_i.

Overflow Protection

Overflow is defined as loss of significance—that is, a computation resulting in a value that cannot be represented in the range of a specific fixed-point type. Overflow refers to a value larger than the largest number in the type range, and underflow refers to a value smaller than the smallest number in the type range.

For the standard Ada fixed-point types, if an overflow occurs, an exception is raised. AutoCode/Ada provides the capability to detect and correct an overflow condition in a fixed-point computation. This is accomplished within the implementations of the generic functions in the SA_FIXED_GENERICS package.

The implementation provided for each of the generic functions uses exception handlers to detect when an overflow occurs. Correction is performed by examining the values of the function and replacing the overflowed value with the extreme value for the data type; the largest number if an overflow, the smallest number if an underflow.

The reliance upon exceptions to detect overflow conditions can be computationally expensive as it may require a significant number of machine instructions to handle the exception. As a result, execution times can suffer. However, if a particular data type used in a calculation is

frequently overflowing, a different data type should be selected. If you are concerned about performance and the use of an exception handler for detecting overflow, the generic functions can be changed. These changes do not affect the generated code in any way. Thus, you can freely modify the generic function implementations without having to regenerate the model.

Caution The provided implementations of the generic functions implement a behavior that is numerically correct and matches the SystemBuild Simulator's results. Any change to the generic functions can severely affect the numeric results of a model. Do not attempt to change the implementation unless you are fully aware of the intricacies of fixed-point numerics.

Stand-Alone Files

Support for the AutoCode/Ada Fixed-Point architecture is found within files in the System-Specific Files `src` directories. Table 3-7 contains all of the fixed-point specific files. Notice that the `.a` extension depends on the Ada compiler you use.

Table 3-7. Fixed-Point Stand-Alone Files

File Name	Package Name	Description
`sa_fxptypes_.a`	`SA_FIXED`	Contains all fixed-point type declarations.
`sa_fxpgenerics_.a`	`SA_FIXED_GENERICS`	Specification of the generic function.
`sa_fxpgenerics.a`	`SA_FIXED_GENERICS`	Package body that implements the generic functions.
`sa_fxpbit_.a`	`SA_FIXED_BITWISE_FUNCTIONS`	Package specification containing bitwise operations on radix 0 fixed-point types.
`sa_fxpbit.a`	`SA_FIXED_BITWISE_FUNCTIONS`	Package body of the bitwise operations on fixed-point types.

Compilation Example

This section illustrates the steps required to generate and compile a model that includes fixed-point types. For the purpose of this example, assume a top-level superblock name of `gaintest`.

1. **Build a model**—Use the SuperBlock Editor to construct a model that uses fixed-point types for input/output signals. Refer to the *SystemBuild User Guide* for more information.
2. **Generate real-time code**—From the **SystemBuild** pull-down menus, select **Build»Generate Real-Time Code**. In the AutoCode dialog box, select **Ada Code Generation** and continue. Example 3-2 shows a sample transcript that should appear in the Xmath window during code generation.

Example 3-2 Example Code Generation Transcript

```
Analyze complete for SuperBlock gaintest.
New Real Time File saved to /home/test/gaintest.rtf
*******************************************************
* AutoCode/Ada Code Generator 7.X                     *
*                                                     *
* (C) Copyright 2000.  National Instruments Corporation.*
* Unpublished work; Restricted rights apply.          *
* All rights reserved.  Portions U.S. Patent.         *
*******************************************************
Loading 'gaintest.rtf' ...
Initializing ...
Building symbols ...
Executing 'ada_rt.dac' :
              Generating procedures package declarations ...
              Generating subsystems package declarations ...
              Generating the scheduler ...
              Generating subsystems package bodies ...
              Generating procedures package bodies ...
Output generated in gaintest.a

              Generating fixed point operators package specification ...
                   Generating operator instantiations ...
                   Generating conversion instantiations ...
Output generated in fxp_gaintest_.a

Code generation complete
```

3. **Compile the stand-alone file**s—Before compiling the generated Ada code, all of the stand-alone files must be compiled into the Ada Library. Sample scripts are provided to create an Ada Library in the current working directory and compile all of the stand-alone files into it. The stand-alone files need only be compiled once into the Ada Library.

4. **Compile the generated files**—Two source files are generated, `gaintest.a` and `fxp_gaintest_.a`, as shown in Figure 3-1. The imposed package dependencies (refer to the *Package Dependencies* section) require that the `RT_FIXED_OPERATORS` package be compiled into the Ada Library before the code that represents the model. Thus, the file `fxp_gaintest.a` is compiled before `gaintest.a`.
5. **Create an executable**—This is the final step and it creates an executable file. Refer to your Ada compiler documentation on how to complete this step.

Fixed-Point Type Declarations

Within the `SA_FIXED` package, all of the supported fixed-point data type are declared. Table 3-8 summarizes the fixed-point type specifications.

Table 3-8. Fixed-Point Data Type Summary

Name[1]	Radix Range	Delta[2]	Range[2] (smallest .. largest)
RT_SBYTE*xx*	–16..48	2.0 ** (–*r*)	–(2.0 ** 7 – r)..((2.0 ** 7 – r) – (2.0 ** (–r)))
RT_UBYTE*xx*	–16..48	2.0 ** (–*r*)	0.0 .. ((2.0 ** 8 – *r*) – (2.0 ** (–r)))
RT_SSHORT*xx*	–16..48	2.0 ** (–*r*)	–(2.0 ** 15 – *r*)..((2.0 ** 15 – *r*) – (2.0 ** (–r)))
RT_USHORT*xx*	–16..48	2.0 ** (–*r*)	0.0 .. ((2.0 ** 16 – *r*) – (2.0 ** (–r)))
RT_SLONG*xx*	–16..48	2.0 ** (–*r*)	–(2.0 ** 31 – *r*)..((2.0 ** 31 – *r*) – (2.0 ** (–r)))
RT_ULONG*xx*[3]	–16..48	2.0 ** (–*r*)	0.0 .. ((2.0 ** 31 – *r*) – (2.0 ** (–r)))

[1] *xx* denotes a two-digit number representing the radix, like 03 or 12.

[2] *r* denotes a specific radix.

[3] Ada compiler limitations restrict the range of the data types to be the same as if the data types were signed.

Generic Functions

The generic functions that are used to instantiate overloaded operators and other functions are found in the `SA_FIXED_GENERICS` package, which is found in the `sa_fxpgenerics_.a` and `sa_fxpgenerics.a` files. Table 3-9 summarizes the functions and their purpose. Refer to the package specification for more details.

Table 3-9. Generic Function Summary

Function Name	Purpose
FIXED_ADD	Addition of two fixed-point values.
FIXED_SUB	Subtraction of two fixed-point values.
FIXED_MUL	Multiplication of two fixed-point values.
FIXED_DIV	Division of two fixed-point values.
FIXED_IDENTITY	The identity property of a fixed-point value.
SIGNEDNEGATION	Negation for a value of a signed fixed-point type.
UNSIGNEDNEGATION	Negation for a value of a unsigned fixed-point type.
SIGNEDABS	Absolute value for a value of a signed fixed-point type.
UNSIGNEDABS	Absolute value for a value of a unsigned fixed-point type.
LESSTHAN	Tests less-than relation between values of two different fixed-point types.
GREATERTHAN	Tests greater-than relation between values of two different fixed-point types.
LESSEQUAL	Tests less-than or equal to relation between values of two different fixed-point types.
GREATEREQUAL	Tests greater-than-or-equal-to relation between values of two different fixed-point types.
INTCAST	Fixed-point value to RT_INTEGER conversion.
INTCAST_TRUNC	Fixed-point value to RT_INTEGER conversion using truncation.
INTCAST_ROUND	Fixed-point value to RT_INTEGER conversion using rounding.

Table 3-9. Generic Function Summary (Continued)

Function Name	Purpose
LONGINTCAST	Fixed-point value to RT_LONG_INTEGER conversion.
LONGINTCAST_TRUNC	Fixed-point value to RT_LONG_INTEGER conversion using truncation.
LONGINTCAST_ROUND	Fixed-point value to RT_LONG_INTEGER conversion using rounding.
BOOLEANCAST	Fixed-point value to RT_BOOLEAN conversion.
INTFIXEDCAST	RT_INTEGER to a fixed-point type conversion.
LONGINTFIXEDCAST	RT_LONG_INTEGER to a fixed-point type conversion.
BOOLEANFIXEDCAST	RT_BOOLEAN to a fixed-point type conversion.
FLOATFIXEDCAST	RT_FLOAT to a fixed-point type conversion.
FLOATFIXEDCAST_TRUNC	RT_FLOAT to a fixed-point type conversion with truncation.
FLOATFIXEDCAST_ROUND	RT_FLOAT to a fixed-point type conversion with rounding.
FIXED_CAST	Conversion between two different fixed-point types.
FIXED_CAST_TRUNC	Conversion between two different fixed-point types with truncation.
FIXED_CAST_ROUND	Conversion between two different fixed-point types with rounding.
NOOPCAST	Conversion to the same fixed-point type.

Bit-Wise Functions

A restricted set of bit-wise operations have been defined for certain fixed-point types. These functions exist in the SA_FIXED_BITWISE_FUNCTIONS package found in the sa_fxpbit_.a and sa_fxpbit.a files. The set of bit-wise operations are the following three functions: BTEST, BCLEAR, and BSET. BTEST tests the *n*th bit of a value. BCLEAR clears the *n*th bit of a value. BSET sets the *n*th bit of a value. The functions are overloaded to accept values that are of a fixed-point type with radix 0. That includes RT_SBYTE, RT_UBYTE, RT_SSHORT, RT_USHORT, RT_SLONG, and RT_ULONG. Bit-wise operations on fixed-point data types can only be done using variable blocks.

Note The Ada templates provided by multiprocessor do not generate code that WITHs or USEs the SA_FIXED_BITWISE_FUNCTIONS package. If you decide to use any of those functions, you must modify the template to include both the WITH and USE clauses for this package. Refer to the *Template Programming Language User Guide* for information about the templates.

Instantiated Functions Package

AutoCode/Ada will generate one additional file for a model if it contains any fixed-point type. This file contains the package for the RT_FIXED_OPERATORS package. This package contains all of the instantiations for every overload operator and conversion function that are used in the model.

Note All of the instantiated functions should have an Ada pragma directive to inline the function. This eliminates the function call overhead associated with overloading operators. Due to some Ada compiler inconsistencies, these directives are commented out of the generated package. A small modification to the template can enable these directives. Refer to the *Template Programming Language User Guide* for information about the templates.

Operator Instantiations

The RT_FIXED_OPERATORS package contains instantiations for overloaded operators. These include the Ada operators: +, -, *, /, <, <=, >, >=, ABS. The appropriate generic function from the SA_FIXED_GENERICS package is chosen to instantiate the overload. Relational operators can be optimized if the data types of both arguments are the same. In that case, a new function is not instantiated and the predefined relational operator is renamed so that it is visible.

Conversion Function Instantiations

The RT_FIXED_OPERATORS package contains instantiations of functions that represent conversion of values to and from a fixed-point type. The appropriate generic function from the SA_FIXED_GENERICS package is chosen to instantiate the function. The name of the instantiated function follows a convention that indicates what type of conversion is to be done. Table 3-10 defines the naming convention for conversion functions.

Table 3-10. Conversion Function Naming Conventions

Conversion Type	Name[1]
Fixed to RT_INTEGER	RT_*YYxx*_I
Fixed to RT_INTEGER (truncation)	RT_*YYxx*_It
Fixed to RT_INTEGER (round)	RT_*YYxx*_Ir
Fixed to RT_LONG_INTEGER	RT_*YYxx*_LI
Fixed to RT_LONG_INTEGER (truncation)	RT_*YYxx*_LIt
Fixed to RT_LONG_INTEGER (round)	RT_*YYxx*_LIr
Fixed to RT_BOOLEAN	RT_*YYxx*_B
Fixed to RT_FLOAT	RT_FLOAT
RT_INTEGER to Fixed	RT_*YYxx*
RT_LONG_INTEGER to Fixed	RT_*YYxx*
RT_BOOLEAN to Fixed	RT_*YYxx*
RT_FLOAT to Fixed	RT_*YYxx*
RT_FLOAT to Fixed (truncation)	RT_*YYxx*t
RT_FLOAT to Fixed (round)	RT_*YYxx*r
Fixed to Fixed (different types)	RT_*YYxx*
Fixed to Fixed (different types, truncation)	RT_*YYxx*t
Fixed to Fixed (different types, round)	RT_*YYxx*r
Fixed to Fixed (same type, no-op cast)	YYxx

[1] *YY* indicates the short name of a fixed-point type *xx* is the two-digit radix.

Sample Package

Example 3-3 shows a generated RT_FIXED_OPERATORS package.

Example 3-3 Generated RT_FIXED_OPERATORS Package

```
--------------------------------------------------------------------------
--                  AutoCode/Ada (TM) Code Generator 7.X                 --
--            National Instruments Corporation, Austin, Texas            --
--------------------------------------------------------------------------
-- rtf filename           : feed.rtf
-- Filename               : fxp_feed_.a
-- Dac filename           : ada_sim.dac
-- Generated on           : Fri Jun  2 14:44:02 1999
-- Dac file created on    : Thu Jun  1 16:19:31 1999
--
--             Fixed Point Operator Instantiation Package
--
--
with SA_TYPES;
with SA_FIXED;
with SA_FIXED_GENERICS;
package RT_FIXED_OPERATORS is
    -- Operator Instantiations --
    function "+" is new SA_FIXED_GENERICS.FIXED_ADD(
                                        SA_FIXED.RT_SSHORT15,
                                        SA_FIXED.RT_SSHORT15,
                                        SA_FIXED.RT_SSHORT15,
                                        SA_FIXED.RT_SLONG15);
    --pragma inline ("+");
    function "-" is new SA_FIXED_GENERICS.FIXED_SUB(
                                        SA_FIXED.RT_USHORT13,
                                        SA_FIXED.RT_USHORT15,
                                        SA_FIXED.RT_SSHORT12,
                                        SA_FIXED.RT_ULONG12);
    --pragma inline ("-");
    function "-" is new SA_FIXED_GENERICS.FIXED_SUB(
                                        SA_FIXED.RT_SSHORT12,
                                        SA_FIXED.RT_SSHORT15,
                                        SA_FIXED.RT_SSHORT14,
                                        SA_FIXED.RT_SLONG14);
    --pragma inline ("-");
```

```
function ">=" is new SA_FIXED_GENERICS.GREATEREQUAL(SA_FIXED.RT_SSHORT14,
                                                    SA_FIXED.RT_SSHORT08);
    --pragma inline (">=");
    function ">=" (LEFT, RIGHT : SA_FIXED.RT_SSHORT13) return BOOLEAN
                                                    renames SA_FIXED.">=";
    --pragma inline (">=");
    function RT_US11 is new
SA_FIXED_GENERICS.FLOATFIXEDCAST(SA_FIXED.RT_USHORT11);
    --pragma inline (RT_US11);
     -- Conversion Function Instantiations --
    function RT_US13r is new
SA_FIXED_GENERICS.FLOATFIXEDCAST_ROUND(SA_FIXED.RT_USHORT13);
    --pragma inline (RT_US13r);
    function RT_SS12_It is new
SA_FIXED_GENERICS.INTCAST_TRUNC(SA_FIXED.RT_SSHORT12);
    --pragma inline (RT_SS12_It);
end RT_FIXED_OPERATORS;
```

Addition and Subtraction Functions

The FIXED_ADD and FIXED_SUB generic functions implement addition and subtraction of fixed-point types. Unlike the predefined Ada fixed-point operators, these generics support mixed-type operators, that is, the types of the operands do not have to be the same. To achieve results as accurate as possible without introducing overflow requires the use of an intermediate type in the calculation. The intermediate type is chosen such that the following properties are maintained: the values of each operand do not overflow when converted to the intermediate type; the result of the operation does not overflow when represented in the intermediate type. If such an intermediate type can be found for the two operands of the operation, the result is guaranteed to be exact. Therefore, addition and subtraction uses the intermediate type for the calculation of the result such that the operations are defined as:

```
c = a + b ==> c = RESULT_TYPE( IT(a) + IT(b) )
c = a - b ==> c = RESULT_TYPE( IT(a) - IT(b) )
```

The two operands are converted to the intermediate type (IT), and then the operation is performed. Then, the result is converted to the result type (RESULT_TYPE). Loss of significance can occur when converting to the result type.

The selection of the intermediate type is performed by the code generator. The selection involves a set of rules that rely upon word size extension. Word size extension is selecting a fixed-point type with a larger number of bits that can be used to represent model numbers. Also, the radix of the intermediate type is chosen to be the radix of the result fixed-point type. For all combinations of all the `RT_SBYTE`, `RT_UBYTE`, `RT_SSHORT` and `RT_USHORT` types, word size extension is possible. However, if any of the `RT_SLONG` or `RT_ULONG` fixed-point types is one of the operator's operands, word size extension is not possible because there are no 64-bit fixed-point types. Example 3-4 and Example 3-5 illustrate that accuracy is maximized when a word-sized extended intermediate type is used in the calculation.

Example 3-4 Word Size Extended Intermediate Type Subtraction Example

Given: n1 is an `RT_SBYTE04`, n2 is an `RT_SBYTE05` and n3 is an `RT_SBYTE07`.

n1 = 1.0625, n2 = 1.0, perform n3 = n1 – n2.

Select *intermediate type* of `RT_SSHORT07` and convert n1 and n2 to that type resulting in n1a = 1.0625 and n2a = 1.0.

Perform t = n1a – n2a = 0.0625.

Assign n3 = t, performing a conversion from `RT_SSHORT07` to `RT_SBYTE07` resulting in n3 = 0.0625

Example 3-5 Result Type As Intermediate Type Subtraction Example

Given: n1 is an `RT_SBYTE04`, n2 is an `RT_SBYTE05` and n3 is an `RT_SBYTE07`.

n1 = 1.0625, n2 = 1.0, perform n3 = n1 – 0n2.

Convert n1 and n2 to the type of n3, `RT_SBYTE07`. Both values of n1(1.0625) and n2(1.0) are not model numbers of the `RT_SBYTE07` type, thus both overflow when converted. The largest model number is substituted so that n1a = 0.9921875 and n2a = 0.9921875.

Perform n3 = n1a – n2a = 0.0

Note The type that is to be used as the intermediate type is represented as a formal parameter to the addition and subtraction generic functions. There is no requirement that the implementation of the function use the intermediate type in the calculation of the result. However, the default implementations do use the intermediate type.

Multiplication and Division Functions

The predefined multiplication and division operators for fixed-point type based arguments are defined in Ada for any combination of fixed-point arguments. The result of the computation is exact and of arbitrary accuracy. Thus, a conversion to the result type of the expression must be performed. During this conversion, accuracy might be lost. The implementation of the generic functions that perform multiplication and division use the predefined operators and perform the conversion to the result type.

32-Bit Multiplication

Multiplication of two 32-bit fixed-point numbers might not necessarily be exact. The problem is that the predefined operator is sometimes unable to use an extended 64-bit calculation to perform the operation. Thus, the result might not be exact. As a rule of thumb, if the sum of the radices of the types of the operands is less than 31, the result should be exact. If that sum is larger than 31, loss of precision might occur.

Note Computation of 32-bit values is compiler vendor dependent. Results compared against the equivalent floating-point computation can vary significantly. The only solution is to upgrade to a version of an Ada compiler that implements more robust fixed-point numerics.

32-Bit Division

Division of two 32-bit fixed-point numbers might not be exact. The problem is that the predefined operator is sometimes unable to use an extended 64-bit calculation to perform the operation. Thus, the result might not be exact. As a rule of thumb, if the radix of the type of the denominator is less than 16, the result should be exact; otherwise, loss of precision might occur.

Conversion Functions

Values from one data type (fixed-point or other type) might need to be converted to another data type (fixed-point or other type). For any conversion of a value that has type of lesser accuracy to a type with a greater accuracy, loss of precision will not occur, but overflow is possible. For example, a value represented in `RT_SSHORT01` converted to `RT_SSHORT03` will not lose accuracy, unless the value overflows. However, when converting a value that has a type of greater accuracy to a type that has a lesser accuracy, loss of precision will occur, but there is

no chance of overflow. To support these issues there are three types of conversion functions:

- Language-defined conversion
- Truncation conversion
- Explicit rounding conversion

These conversions are described in the following sections.

Language-Defined Conversion

The Ada language provides four data type conversions. The rules in Ada that govern the conversion are explicit except for one detail. When converting between different numeric types, values that cannot be exactly represented in the type are rounded to the nearest model number. A value that is exactly halfway between two model numbers (that is, at the midpoint), can be rounded to the larger or smaller model number. The choice is left to the implementor of the Ada compiler. Instantiated conversion functions that use the language-defined conversion do not have a *t* or *r* designator at the end of the function name. Refer to Table 3-10.

Truncation Conversion

This type of conversion implements *truncation* for values that are at the midpoint between two model numbers. For example, if a value of 1.5 in the `RT_SBYTE01` type is converted to `RT_SBYTE`, the resulting value is 1.0. Instantiated functions that implement truncation have a *t* designator at the end of the function name. Also, generic functions that use truncation have the `_TRUNC` suffix as part of those names.

Explicit Rounding Conversion

This type of conversion implements *round away from zero* rounding mode for values that are at the midpoint between two model numbers. For example, if a value of 1.5 in the `RT_SBYTE01` type is converted to `RT_SBYTE`, the resulting value is 2.0. Instantiated functions that implement this rounding mode have an *r* designator at the end of the function name. Also, generic functions that use rounding have the `_ROUND` suffix to those names.

The choice of which type of conversion function to use depends on the situation. For example, the Signal Conversion block can use either truncation or rounding. However, when dealing with fixed-point numerics, there are many other implicit conversions between data types, such as the conversion of arguments to an intermediate type for addition or subtraction.

For these types of conversions, the language-defined conversion is used. In general, when an explicit conversion is required and there is no specification of which to choose, AutoCode/Ada will select the explicit rounding conversion.

Using System-Level Parameters to Generate Instantiations

Before you can use the system-level parameters to generate operator or conversion instantiations, all of the subsystems and procedures must be *generated*. Simply scoping to a subsystem or procedure is not sufficient. Code must be generated so that the exact operators and conversion used in the subsystem or procedure can be determined and recorded.

After all of the subsystems and procedures are generated, you must call the `tpl` function `collect_fxpdata( )`. Until this function is called, all of the `fxp` parameters are empty. After being called, all of the operators and conversions used in the whole model are placed into the system-level `fxp` parameters. No assumptions can be made about the order *within* a given list parameter. However, the *n*th element in one list does relate to the *n*th element in another list, depending on the purpose of the lists.

For the `fxp_operatorid_li` and `fxp_conversionid_li` parameters, these contain the type of operator or conversion to be instantiated. However, the lists might contain duplicates. An operator or conversion instantiation cannot be declared twice in the package specification. Thus, duplicate operators or conversions must not be generated. Therefore, any operator or conversion identifier with a value of 1,000 or larger must not be instantiated.

In addition to containing all of the operators and conversion for all of the subsystems and procedures in a model, the list can contain other operators or conversions that are used in the standard packages, such as the scheduler or system data package.

Using Subsystem-Level Parameters to Generate Instantiations

Before you can use the subsystem/procedure-level parameters to generate operator or conversion instantiations, all of the subsystems and procedures must be *code generated*. Simply scoping to a subsystem or procedure is not sufficient. Code must be generated so the exact operators and conversion used in the subsystem or procedure can be determined and recorded.

After all of the subsystems and procedures are generated, you must call the `tpl` function `collect_fxpdata( )`. Until this function is called and a subsystem or procedure is scoped, all of the `sp_fxp` parameters are empty.

The data in the sp_fxp parameters represent the operators and conversion used in the currently scoped subsystem or procedure. No assumptions can be made about the order within a given list parameter. However, the *n*th element in one list does relate to the *n*th element in another list, depending on the purpose of the lists.

For the sp_fxp_operatorid_li and sp_fxp_conversionid_li parameters, these contain the type of operator or conversion to be instantiated. The lists can indicate duplicates with respect to all of the operators or conversion in the model. Therefore, operators that are indicated to be duplicate are not duplicate with respect to the currently scoped subsystem or procedure. So, that operator must be instantiated. This is done by subtracting 1,000 from the value and continuing processing as normal.

You should loop through all of the subsystems and procedures of a model, scope it, then generate the operator and conversion instantiations. It is implied that different packages must be used for each subsystem and procedure as there might be duplicate operators or procedures between subsystems and procedures.

System Scope Operators and Conversions

A so-called system scope exists for operators and conversions that are used somewhere other than a subsystem or procedure. When using the subsystem-level parameters to generate instantiations, you must also generate instantiations for these system scope operators and conversions. Unfortunately, the sp_fxp parameters are not available when scoped to just a system. Therefore, you must use the system-level parameters to generate these instantiations.

The system-level parameters will contain all of the operators and conversions in some order. The only guarantee is that the system scoped operators and conversions appear in the system-level parameter lists after all of the subsystem and procedure operators and conversions. Thus, to find the beginning of the system scoped operators and conversions, you must compute the sum of all of the subsystem and procedure operators, then the sum of all of the subsystem and procedure conversions. These two sums represent the starting index of the system scoped operators and conversions. The total number of system scoped operators or conversions must be computed as the difference between the total number of operators or conversions using the system-level parameters and the computed sum of the number of subsystem and procedure operators or conversions using the subsystem-level parameters. For an example, refer to the ada_fxpt_sub.tpl template.

Known Ada Compiler Problems

The architecture of AutoCode/Ada Fixed-Point heavily relies upon overloaded operators and generic function instantiation. For a large and complex model, the number of overloads and instantiations might overwhelm some older Ada compilers. Also, problems might occur when compiling source code from many different models into the same Ada Library. NI suggests that code from different models be compiled into separate libraries.

Another problem area might be the declaration of unsigned fixed-point types. The Ada'83 standard does not include unsigned data types. Newer Ada compilers do support unsigned types as extensions to the language definition. If your compiler fails to handle unsigned fixed-point types, one solution is to avoid using unsigned fixed-point types in the model. Or, upgrade to a newer Ada compiler.

Another problem might occur with a `pragma` directive. NI strongly suggests that the overloaded operators and conversion functions be specified with the inline directive. This eliminates function call overhead. However, because those functions are instantiated from generics, older Ada compilers might not work properly.

Comparing Results to SystemBuild's Simulator

The SystemBuild Simulator can simulate a model with fixed-point types. If you compare the stand-alone simulations from an Ada executable, the results might not match. The following examples are possible reasons and solutions for the problem:

- **Round and Truncation**—By default, the SystemBuild Simulator performs fixed-point data conversions using truncation as the rounding mode. The Ada language always uses some type of rounding mode *other* than truncation. SystemBuild includes a special default parameter, `fixpt_round`, which when set to the value of 1, uses a *round to nearest*, with midpoint rounded away from zero mode.
- **Different Rounding Modes at Midpoint**—The Ada language specifically states rounding mode is to nearest. However, the specification does not specify rounding when at a midpoint. Table 3-1 shows the possible choices. SystemBuild uses a round to nearest and away from zero at midpoint. Therefore, if your Ada compiler does not round away from zero at midpoint, the results from the simulator and the stand-alone simulation will differ. There is no workaround. The new Ada standard, Ada'95 specifies round away from zero at midpoint.

Table 3-11. Possible Midpoint Round Modes

Mode	INTEGER(5.5)	INTEGER(–6.5)
Toward 0	5	–6
Away from 0	6	–7
Positive Infinity	6	–6
Negative Infinity	5	–7
To Odd	5	–7
To Even	6	–6

- **Floating-Point Textual Representation**—The values generated from a stand-alone simulation are converted to a textual (ASCII) representation. That representation in textual form might not quite be as accurate as possible. That is, the last few digits in the mantissa might be different than the simulator's textual representation. There is no solution other than to upgrade and/or change to a compiler that converts floating-point numbers to a textual form more accurately.
- **32-bit Computations**—Ada specifies that all fixed-point calculations are exact. However, for 32-bit multiplication and division, there might be differences in the algorithm used by the Ada compiler vendor. These differences could affect accuracy when compared against the SystemBuild Simulator. The simulator uses a 64-bit extended integer calculation for 32-bit multiplication and division. Therefore, differences could be a result of 32-bit algorithm differences. It might be necessary to implement your own 32-bit multiplication or division algorithm if the error in the predefined Ada algorithm is too large.

No-Op Conversion Function

The purpose of the so-called no-op conversion function is to provide a hint to the compiler to select the proper overloaded operator. Without such a hint, there can be situations where the selection of the appropriate overloaded operator is ambiguous. Simple expressions of the form `a = b` *`op`* `c` cannot be ambiguous. However, if `a`, `b`, or `c` is a complex subexpression, like `a = b` *`op`* `c` *`op`* `d`, it is possible that there is insufficient type information to resolve which operator is to be used. The code in Example 3-6 illustrates the problem and solution.

Example 3-6 Example Code Causing Ambiguous Selection of Overloaded Operators

```
function "*" (left:SB0; right:SB1) return SL0;
function "*" (left:SB0; right:SB1) return SS0;
function "*" (left:SL0; right:SL0) return SL0;
function "*" (left:SS0; right:SL0) return SL0;
function TO_SLO (left:SL0) return SL0;
V1 : SB0;  V2 : SB1:  VL:SL0;
begin
   VL := V1 * V2 * VL           --ambiguous
   VL := TO_SL0(V1 * V2) * VL;  --unambiguous expression
end;
```

The first assignment is ambiguous because there is more than one choice of the overloaded operator function that would satisfy the first multiplication subexpression. The second assignment is not ambiguous because the `TO_SLO` function is unary and not overloaded. Therefore, the unary function is forcing the `V1*V2` subexpression to return a SL0 result. Thus, there is only one operator that satisfies the second example.

4

Generating Code for Real-Time Operating Systems

This chapter describes the RTOS configuration file and functionality provided for generating code for real-time operating systems.

Real-Time Operating System Configuration File

Code that is to execute under the control of a real-time operating system (RTOS) usually has configuration elements specific to the real-time environment that are not required for stand-alone code. While it is possible to modify the template to configure aspects of the RTOS for a particular model, it is more likely that each model will require different configuration and will commonly undergo a tuning phase during the development process. Instead of directly modifying the template, AutoCode includes the concept of the RTOS configuration file.

The RTOS configuration file is a text file containing tables of information related to the generated source code components of a model, like subsystems and non-scheduled procedure SuperBlocks. Each table contains configuration information about its respective component. Thus, instead of modifying the template code to reconfigure RTOS information, you can modify the values in the RTOS configuration file and then regenerate code. The tables include common aspects related to generating code for execution under a real-time operating system. The template programmer is free to define the meaning of the table data in any way; however, the table names and the template parameter names that contain the data imply our semantic intent for the use of that data. In the remainder of this section, we describe the RTOS configuration tables with a focus on our intended usage of the data and template parameters.

Note The RTOS file cannot be used with any of the `-pmap`, `-smap`, `-bmap`, `-imap`, or `-prio` options.

Configuration Items

The following is a list of configuration attributes for each type of AutoCode component.

- Number of processors
- Scheduler priority
- Subsystem Tasks
  - Priority
  - Stack Size
  - Processor
  - Mode Flags
- Interrupt Procedure SuperBlocks
  - Priority
  - Stack Size
  - Processor
  - Vector
  - Mode Flags
- Background Procedure SuperBlocks
  - Priority
  - Stack Size
  - Processor
  - Ordering
  - Mode Flags
- Startup Procedure SuperBlocks
- Processor
  - Ordering
- Processor
  - IP Number

Table Syntax

This section provides information about the RTOS configuration tables.

Table Naming Convention

Tables are given a name to identify the contents of the data contained therein. Table names are specified in the same form as Xmath variables, `partition.name`. RTOS does not look at the `partition` specifier. Notice that the `name` specifier is case sensitive.

Table Column Contents

The contents of a column within a table has a specific data type—for example, an integer or floating point or string value. For each row in a table, all columns must contain data of the appropriate type. If a table contains no rows, it will not appear in the configuration file.

Table Orderings

The tables can appear in any order in the file except the Version Table, which must be the first table. Also, a table can appear only once in the file. If the table appears more than once, the data in the first table is used.

File Comments

A line starting with the pound character # denotes the line is a comment.

RTOS Configuration File Contents

The RTOS configuration file consists of named tables of data. Some tables are single element, while others have one or more columns of data. The table formats are based on the way Xmath outputs PDM matrix information into a text file.

Processors Table

Table 4-1 is a single element table consisting of the number of processors to target for the generated code. The table is named `processors`. Example 4-1 shows an example of a processors table.

Table 4-1. Processors Table Contents

Column Name	Data Type	Template Parameter Containing the Data	Default Value
N/A	Integer	`nprocessors_i`	Value used with `-np` option or 1

Example 4-1 Processors Table Example

```
rtos.processors =
        2
```

Scheduler Priority Table

Table 4-2 is a single element table consisting of the priority assigned to the scheduler task. The table is named: scheduler_priority. Example 4-2 shows an example of a scheduler priority table.

Table 4-2. Scheduler Priority Table Contents

Column Name	Data Type	Template Parameter Containing the Data	Default Value
N/A	Integer	`scheduler_priority_i`	`ntasks+nintrprocs+2`

Example 4-2 Scheduler Priority Table Example

```
rtos.scheduler_priority =
        254
```

Subsystem Table

Table 4-3 lists the configuration information for all subsystem tasks of the model. Each row is identified with the subsystem task number. The table is named `subsys`. Example 4-3 shows a subsystem table.

Table 4-3. Subsystem Table Contents

Column Name	Data Type	Template Parameter Containing the Data	Default Value
Priority	Integer	`sspriority_li[]`	`scheduler_priority - nintrprocs - n.`
Stack Size	Integer	`ssstack_size_li[]`	`1024`
Processor	Integer	`ssprocessor_map_li[]`	`Round-robin` (a sequential, cyclical allocation of resources to the subsystems)
Mode Flags	String	`ssmode_flags_ls[]`	None

Example 4-3 Subsystem Table Example

```
rtos.subsys =
Subsystem    |        Priority      Stack Size   Processor Mode Flags
-------------------------------------------------------------------
1            |        200           4096         1         K_NOFATAL
2            |        150           4096         1         K_IO | K_ER
3            |        150           8192         2         K_STO
```

Caution The SystemBuild Analyzer identifies how many subsystems a particular model has. AutoCode is limited to providing you with a warning if the subsystems change. AutoCode can detect if new subsystems are added, but not when they are deleted. For example, assume a model and RTOS file for subsystems 1, 2, and 3. If subsystem 2 is deleted and code regenerated using the previous RTOS file, the new subsystem 1 will use subsystem 1's configuration. The new subsystem 2 (which was the old subsystem 3) will use subsystem 2's configuration. Then, AutoCode will report that subsystem 3's data is unused.

Interrupt Procedure SuperBlock Table

Table 4-4 consists of configuration information for all interrupt procedure SuperBlocks in the model. Each row is identified by the name of the SuperBlock. The table is named `intrsupblk`. Example 4-4 shows an interrupt table.

Table 4-4. Interrupt Table Contents

Column Name	Data Type	Template Parameter Containing the Data	Default Value
Priority	Integer	`proc_priority_li[]`	`scheduler_priority - n`
Stack Size	Integer	`proc_stack_size_li[]`	`1024`
Processor	Integer	`proc_processor_map_li[]`	`Round-robin` (a sequential, cyclical allocation of resources to the interrupt SuperBlocks)
Vector	Integer	`proc_vector_li[]`	0
Mode Flags	String	`proc_mode_flags_ls[]`	None

Example 4-4 Interrupt Table Example

```
rtos.intrsupblk =
Interrupt    |     Priority   Stack Size  Processor Vector  Mode Flags
-------------------------------------------------------------------
catch_it     |     200        4096        1         255     NONE
sigio_io     |     150        4096        1         127     I_IO
keypad       |     150        8192        2         31      NONE
```

Background Procedure SuperBlock Table

Table 4-5 consists of configuration information for all background procedure SuperBlocks in the model. Each row is identified by the name of the SuperBlock. The table is named `bkgdsupblk`. Example 4-5 shows a background table.

Table 4-5. Background Table Contents

Column Name	Data Type	Template Parameter Containing the Data	Default Value
Priority	Integer	`proc_priority_li[]`	`scheduler_priority-ntasks - nintrprocs - 1`
Stack Size	Integer	`proc_stack_size_li[]`	`1024`
Processor	Integer	`proc_processor_map_li[]`	`Round-robin` (a sequential, cyclical allocation of resources to the background procedure SuperBlocks)
Ordering	Integer	`proc_ordering_li[]`	$n + 1$
Mode Flags	String	`proc_mode_flags_ls[]`	None

Example 4-5 Background Table Example

```
rtos.bkgdsupblk =
Background   |      Priority  Stack Size  Processor  Ordering   Mode Flags
-------------------------------------------------------------------------
back_1       |     200        4096        1          2          NONE
back_2       |     150        4096        1          1          NONE
```

Startup Procedure SuperBlock Table

Table 4-6 consists of configuration information for all startup procedure SuperBlocks in the model. Each row is identified by the name of the SuperBlock. The table is named `stupsupblk`. Example 4-6 shows a startup table.

Table 4-6. Startup Table Contents

Column Name	Data Type	Template Parameter Containing the Data	Default Value
Processor	Integer	`proc_processor_map_li[]`	`Round-robin` (a sequential, cyclical allocation of resources to the startup procedure SuperBlocks)
Ordering	Integer	`proc_ordering_li[]`	*n* + 1

Example 4-6 Startup Table Example

```
rtos.stupsupblk =
Startup          |   Processor  Ordering
-----------------------------------------
pre_initialize   |   1          1
var_update       |   1          2
io_init          |   2          1
ipc_init         |   2          2
```

Processor IP Name Table

Table 4-7 consists of configuration information, assigning each processor an IP name. Each row is identified by the processor number. The table is named `IPprsrmap`. Example 4-7 shows a processor IP table.

Table 4-7. Processor IP Table Contents

Column Name	Data Type	Template Parameter Containing the Data	Default Value
IP Number	String	`prsr_ip_name_ls[]`	127.0.0.1

Example 4-7 Processor IP Table Example

```
rtos.IPprsrmap =
Processor |  IP Name
-----------------------------------------------------------------------
1         |  127.0.0.1
2         |  133.65.32.111
```

Version Table

Table 4-8 is a single element table consisting of the version number of the format of the configuration tables. The table is named `version`. Example 4-8 shows a version table.

Table 4-8. Version Table Contents

Column Name	Data Type	Template Parameter Containing the Data	Default Value
N/A	Integer	N/A	Latest file version

Example 4-8 Version Table Example

```
rtos.version =
        1
```

Using the Configuration File

Two mutually exclusive command options control the usage of the RTOS configuration file, `-rtos` and `-rtosf`. The `-rtos` option uses a default filename as the name of the configuration file to read and keep updated. This filename is `ac_rtos.cfg`. Use `-rtos` to create the configuration file the first time. Each time AutoCode is executed with the `-rtos` option, the default configuration file is updated. This update contains all of the configuration data that was used to generate code for the model and any additional AutoCode components—that is, subsystem tasks, nonscheduled SuperBlocks—in the model that did not have data in the configuration file.

The `-rtosf` option requires a filename to be specified as an argument. When this option is used, AutoCode reads data from the file specified, ignoring the default file. However, when data is updated, the new data is stored in the file *`filename`*`.new`, where *`filename`* is the filename specified along with the `-rtosf` option.

If you use an old configuration file with an updated model, AutoCode will report differences and remove any unused configuration data when the

configuration file is updated. Also, the updated configuration file is stored in the same directory where the generated code is placed. Refer to the `-o` option.

Caution Comments are *not* preserved in the default configuration file. Use a different file and the `-rtosf` option if you plan to preserve the comments.

Note You *must* use the template parameters as specified in the previous tables before code affected by RTOS file settings is generated. The standard C and Ada templates do not have any of the RTOS template parameters. It is up to a template programmer to integrate the template parameters in such a way to customize code generation for a particular real-time operating system. Refer to the *Template Programming Language User Guide* for template examples.

5

Generated Code Architecture

This chapter supplies more details about the content and framework of the generated code. This includes storage usage, various procedures, specialized blocks, and subsystems. This chapter is directed toward someone writing his/her own template, interfacing generated code within a larger system, designing for optimal code size and speed, or being interested in the generated code architecture. This chapter assumes familiarity with the C or Ada programming language and standard programming concepts.

Symbolic Name Creation

AutoCode is very much like a compiler in that it translates a model diagram (a type of language) into generated code. The requirements of generated code are somewhat different than that of the model diagram and AutoCode must resolve these differences and preserve the semantics of the model diagram.

The symbolic names that appear within the generated code include function names, variable names, and data types. AutoCode uses the names of SuperBlocks, blocks, and signal labels/names to derive the symbolic names for the code that provide maximal traceability from the code back to the model diagram. AutoCode follows some basic rules when deriving a symbolic name. These rules are described in the following sections.

Default Names

AutoCode selects a default name for any symbol that does not have a specific name from the diagram. A default name is a combination of the block name and block identification number as found in the diagram. If the block has no name, the enclosing SuperBlock name is used to form the name.

Signal Naming

A signal in the diagram is represented as a variable in the generated code. Within the diagram, signals can have two names: a *label* and a *name*. The signal's label can appear in the diagram while a signal's name does not appear. If a signal has both a label and a name, AutoCode uses the name to generate the symbolic name in the generated code. If there is only a label or a name, AutoCode uses whichever one is specified. If you specify neither a label nor a name, then AutoCode uses a default name.

Duplicate Names

There is no restriction in the model diagram about reusing the same names over and over again. However, AutoCode must make sure that the symbolic names in the generated code are unique. AutoCode solves this problem by mangling the name of duplicates, that is, appending a suffix to the original name to preserve uniqueness.

Selection of a Signal Name

In a model diagram, there might be many levels of hierarchy represented as nested SuperBlocks. SuperBlocks of the same rate are combined into subsystems and thus are effectively optimized away. Refer to the *Subsystems* section for more information. Given that, the labels specified at the boundary of SuperBlocks are not used. If they were, there would be extra copies of data because of a name change, which would not represent optimal code generation. Therefore, within a hierarchy of SuperBlocks, the label/name from the block that computes the signal is used as the symbolic name of the variable.

Subsystem and Procedure Boundaries

There are times when the names at the SuperBlock boundary are used. The names are used only when a definitive interface is created by a SuperBlock. This occurs when the SuperBlock represents a subsystem boundary or a Procedure SuperBlock.

Typecheck Feature and Data Types

The Typecheck feature of the SystemBuild Analyzer directly impacts the data types for signals generated by AutoCode. When the feature is enabled, data types that are specified in the model diagram will be used in the generated code. If the feature is disabled, all data types will be considered float.

Global Storage

In a strict modular programming paradigm, global storage is strictly avoided. However, global storage can be used safely and provides significant benefits in terms of code size and speed. Traditionally, AutoCode generates code that enforces a policy of safe access to global data. However, the need for potentially unsafe uses of global memory to achieve tight and efficient production code requires more latitude in the code generation, and requires a lot of effort to design properly. There is the possibility of losing determinacy and reentrancy.

Percent vars (%var)

Percent vars represent parameterized data specified within some SystemBuild Blocks. A %var is generated as a global variable. AutoCode uses a %var variable in a read-only context. If a %var parameter can be written to, as in the case of passing parameters to a UCB, duplicate data is generated to allow the data to change, but at the same time, not affect the read-only uses. Notice that a %var value can be changed as part of a startup procedure SuperBlock. Refer to the *Startup* section for details.

Global Variable Blocks

Global Variable Blocks are implemented as global variables in the generated code. AutoCode sequences and generates code that provides a safe and deterministic use of variables blocks. You must generate variable block callouts to form a critical section to maintain integrity of the global variable blocks when used in a pre-emptive multi-tasking system. This safety can be expensive if performance is an issue and you might not get the behavior you want because of the special sequencing.

Sequencing Variable Blocks

The sequence when a read from and write to a variable block occurs is critical in maintaining determinacy. The following is a summary of when the block will execute.

- **Read-From Variable Block (ALL Addressing mode)**—These blocks execute *before* any other type of block within the subsystem, procedure, or sequence frame. The data from the variable block is copied into local variables. These local variables are accessed in place of the actual global variable.

- **Write-To Variable Block (ALL Addressing mode)**—These blocks execute *after* all other types of blocks within the subsystem, procedure or sequence frame.
- **Read-From/Write-To Variable Block (Element/Bit Addressing modes)**—These variants to access variable block information are sequenced just like any other block; that is, it is sequenced after its inputs have been computed.

Global Variable Block and %var Equivalence

If a Global Variable Block and a %var use the same name, only one variable is generated and that variable will be used for all instances of the %var and Variable Block.

Optimization for Read-From Variable Blocks

AutoCode supports an optimization for Read-From Variable Blocks (Global and Local) that eliminates the copy and directly accesses the local or global variable. This optimization is optional and the optimization of global and local variable blocks can be separately controlled. Global variable block optimization works only if variable blocks are not shared between multiple subsystems and when the `-vbco` option is not used.

Global Scope Signal Capability

The memory and performance requirements of real-time production code force the issue of global variables. AutoCode does not generally use global variables; rather, it creates and uses stack variables and explicit interfaces. While that architecture is perfectly sound, it does cause overhead in a production system. Therefore, AutoCode supports direct use of global variables for local block outputs within a subsystem. Extending that concept allows global variable(s) to be used as the inputs and outputs of a procedure.

Chapter 9, *Global Scope Signals and Parameterless Procedures*, provides information about using global scope signals and using global variables as the input(s) and output(s) of a procedure. Such a procedure is called a parameterless procedure.

Subsystems

This section describes the design and operation of subsystems. This includes:

- *Discrete and Continuous SuperBlocks Versus Subsystems*
- *Block Ordering*
- *Interface Layers*
- *Scheduler External Interface Layer*
- *System External Interface Layer*
- *Discrete Subsystem Interface Layer*
- *Static Data Within Subsystems*
- *Pre-init Phase*
- *Init, Output, and State Phases*
- *Copy Back and Duplicates*
- *Error Handling*

Discrete and Continuous SuperBlocks Versus Subsystems

A SuperBlock is a SuperBlock Editor concept that acts as a container that describes a type and/or timing attributes for a set of blocks. There are various flavors of SuperBlocks, such as Procedure SuperBlocks, which have different code generation impacts. For the moment, the current discussion is limited to Discrete and Continuous SuperBlocks.

The SystemBuild Analyzer, which is an internal thread of the Simulator, translates a model into a representation (`.rtf` file) that AutoCode and other applications can understand. One of the primary tasks of the Analyzer is to create subsystems from SuperBlocks. A subsystem is the set of SuperBlocks with the same timing attributes; discrete SuperBlocks form discrete subsystems while continuous SuperBlocks form a single continuous subsystem. The Analyzer takes the blocks, which the SuperBlocks identified as one subsystem, and creates a block ordering. As a result, AutoCode knows nothing about SuperBlocks and can generate only what the Analyzer partitioned into subsystems.

Top-Level SuperBlock

The term Top-Level SuperBlock is often used. This term describes the SuperBlock that was the root of the Analyzer's processing of the SuperBlock hierarchy into subsystems, that is, a starting point for the translation. AutoCode uses this Top-Level SuperBlock to provide default names for the entry point of the stand-alone simulation, and more importantly defines a boundary to determine which signals are the external inputs and outputs of the system. The Top-Level SuperBlock also gets translated into a subsystem just like all of the other SuperBlocks.

Block Ordering

Another task of the Analyzer is to determine the execution order of the basic blocks within a subsystem. In other words, the Analyzer sorts the blocks. This sequence is controlled by a data-flow analysis of the signal connectivity of the model. Parallel threads of execution can be sorted in any order as long as data-flow integrity is maintained. There are special blocks that have special sequencing requirements (for example,Variable Block) that the Analyzer sorts.

There is a block called the Sequencer Block that provides additional information about how to sort the blocks, which allows large-grain control over sequencing. A Sequencer block divides the diagram into a left-side frame and a right-side frame. Blocks in the left-side frame are guaranteed to execute before the blocks in the right-side frame. Blocks within a frame are sorted based on data-flow and any special rules, just like a subsystem.

Because block ordering is done before AutoCode is invoked, AutoCode cannot change the ordering of the blocks within the generated code.

Interface Layers

There are three layers of interface within the framework generated by AutoCode. The three layers are used to support the following:

- Interfacing to hardware
- Data type support
- Multi-rate subsystem communication

Figure 5-1 illustrates the interface layers. The layers are described in the subsections shown in the figure.

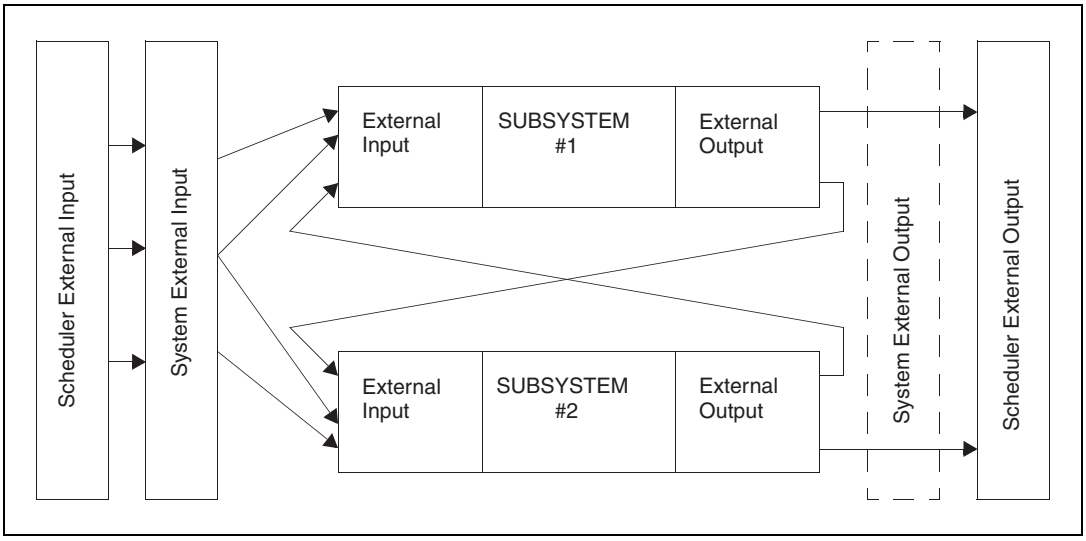


Figure 5-1. Interface Layering Diagram

Scheduler External Interface Layer

This layer refers to the data directly interfacing with hardware or some other entity not modeled within the SystemBuild model. This layer is represented as two arrays of floating-point numbers, one for external inputs (called `ExtIn`) and another for external outputs (called `ExtOut`).

System External Interface Layer

This layer uses two structures, `Sys_ExtIn` and `Sys_ExtOut`, to respectively represent the system external inputs and outputs. Normally, the `Sys_ExtOut` structure is optimized out because the scheduler external outputs are taken from a subset of the subsystem's external outputs. A special quality of the System external structures is that the members of those structures are ordered relative to the Top-Level SuperBlock. In other words, for example, the fifth external input of the Top-Level SuperBlock will appear as the fifth variable within the structure. This layer is needed because this layer represents the actual data types as specified relative to the Top-Level SuperBlock. Therefore, data copied into and out of this layer is subject to a data type conversion into the all-float Scheduler interface.

Discrete Subsystem Interface Layer

This layer comes in two variations to allow for both an optimized and general solution. The external interface to a discrete subsystem is represented by two structures, one representing the subsystem external inputs and the other the subsystem external outputs. These structures are generically referred to as the `U`-structure for the inputs and `Y`-structure for the outputs. Within the subsystem, pointers to the `U` and `Y` structures are defined as formal parameters to the subsystem procedure with the names `U` and `Y`, respectively.

Single-Rate System

A single-rate system, (that is, a system that contains only one subsystem and no disconnected signals) is optimized to eliminate extra copies of the external input/output data from the System interface. Therefore, a single-rate model uses the System external inputs as the subsystem's external inputs, and the subsystem's external outputs as the System's external outputs.

Multi-Rate System

A multi-rate system represents the general case for external input and output data. In a multi-rate system, a particular subsystem output might be used by other subsystems or might be an external system output or some combination. AutoCode minimizes the connections between subsystems and the system external output. To accomplish this, AutoCode must rearrange the ordering of members within the `U` and `Y` structures, as opposed to the order being maintained in the single-rate case.

Sample and Hold

A property of a multi-rate model is the concept of sample and hold. This refers to the capturing (holding) of the input data of a subsystem until it completes. For example, in Figure 5-1, assume subsystem 1 executes at a rate of 0.1 (10 Hz) while subsystem 2 executes at 0.2 (5 Hz). Each subsystem depends on the other for 1 input. Given that subsystem 1 executes twice as fast as subsystem 2, the output of subsystem 1, used by subsystem 2, changes during subsystem 2's execution. This means that if there were no sample and hold, subsystem 2 could potentially compute values using different values at different times during the execution of the subsystem. This non-determinacy can produce unexpected results.

This sample and hold mechanism guarantees deterministic behavior for all possible connectivities and is implemented using a technique called double buffering. Double buffering involves swapping of pointers.

Static Data Within Subsystems

The implementation of blocks within a subsystem might require persistent data. This data can be characterized as look-up tables, parameter information, as well as data needed to manage the scheduling of the subsystem. The use of static data prevents subsystems from being safely used as reentrant code.

iinfo

The `iinfo` is an array that contains flags that indicate the phase and error condition of the subsystem. These are used by the scheduler. The phase might be an initialization, output update, or state computation (INIT, OUTPUT, and STATES, respectively). Error status is checked for during the execution of the subsystem and, if a run-time error was detected, a flag is set.

R_P and I_P

`R_P` and `I_P` are static arrays used to store floating point (real) and integer parameter data, hence their names. Many blocks require static data in the form of look-up tables, hard-coded parameter data, and initial conditions. This type of block-specific data from all blocks within the subsystem are placed within the arrays, depending on the data type of the data. Notice that fixed-point data is stored in separate arrays to maintain the proper data type and eliminate data type conversion. It is important to realize that `R_P` and `I_P` data can change during the execution of a subsystem and that to support a restarting capability on some hardware targets, the initial data is preserved in a second `R_P` and `I_P` array that is only used for initialization. You can disable the restart capability, which potentially can greatly reduce the footprint of the object code.

State Data

Another category of data is specifically data related to the state of a block. State data is differentiated because the semantics of state data are taken from Control-System theory. Many of the standard blocks rely on the semantics of state data to properly implement the block's algorithm. State data is managed within the subsystem and involves swapping pointers representing the current and next states.

Procedure Data

Procedure SuperBlocks have inputs, outputs and states independent of the subsystem from which the procedure is called. This is required to properly implement the characteristics of a procedure; those characteristics include reusability and reentrancy. Therefore, for each instance of a call to a procedure SuperBlock within a subsystem, a set of static variables is created for each of the structures that describe the definition of the procedure.

Pre-init Phase

The purpose of the pre-init phase is to initialize all of the subsystem's static data. This phase is called for each subsystem before the simulation begins, that is, before time = 0.0.

Init, Output, and State Phases

A subsystem has three phases that can be easily seen in the generated code. The code related to these phases is guarded by If-statements or comments. The subsystem can be in multiple phases at the same time.

- **Init Phase**—Is executed only once at the first time the subsystem is called. This might be at time = 0.0 but might be later if the subsystem is skewed, triggered, or enabled. Each block can have code for its initialization. Activities such as setting initial state and output data are typical.
- **Output Phase**—Occurs for each time point for the subsystem. The purpose of this phase is for each block to compute its outputs for the current time point. The output phase always occurs *before* the state phase.
- **State Phase**—Occurs for each time point for a discrete subsystem and can be multiple times for a continuous subsystem. The purpose of this phase is for each block to compute the value of next state to be used by the block at the next time point. This phase always occurs *after* the output phase.

Copy Back and Duplicates

After all of the code for the three phases has had a chance to execute, there might be additional code to perform what is called a copy-back, or to deal with duplicates. This only applies to subsystem external outputs. A copy-back occurs only in vectorizing code in which part of an array is used as a subsystem external output. Duplicates can only occur in a single-rate

system, because ordering of the outputs in a single-rate system is maintained. In a multi-rate system, duplicates can be safely eliminated because of the sample and hold mechanism. The code to perform both of these activities represent a copy from one variable to another.

Error Handling

The error handling within a subsystem is very simple. When an error is detected, the code goes to (or throws an exception) some error handling code. The purpose of the error handler is to cope with the error as best it can. The error handler is completely defined in the template and you can change the default behavior quite easily. There is only a limited set of run-time errors that are checked for, but some blocks generate special error handling when the `-e` option is specified for code generation.

Standard Procedures

A Standard Procedure SuperBlock represents a reusable, reentrant function within the generated code. A Standard Procedure is internally structured almost exactly like a subsystem, except for reusability and reentrancy.

Structure-Based Interface

This form of the interface to the Standard Procedure is conceptually the same as that of the subsystem. The inputs, outputs and other data are packaged into structures that are passed by pointer as actual arguments to the function. The caller is tasked with creating instances of the structures needed for the interface. The following briefly describes the type of structures required for this interface.

- **Input**—Also called the `U`-structure and contains the inputs of the procedure. The inputs are in the order specified in the procedure definition's external inputs.
- **Output**—Also called the `Y`-structure and contains the outputs of the procedure. The outputs are in the pin-order specified by the connections of basic blocks to the external output pins of the procedure definition.
- **Info**—Provides additional information that is used by the procedure to maintain reusability and reentrancy. Data includes the `R_P` and `I_P` data for the blocks within the procedure. State data for the blocks within the procedure appear within this structure as well. The input, output and info structures of nested procedure SuperBlocks appear as well. Also, %vars used in the procedure are passed by pointer to support the partitioning capability of SuperBlocks.

Unrolled Interface

There is another form of the procedural interface, the unrolled interface (`No-UY`). This interface does not use `U`- and `Y`-structures to pass the inputs and outputs. The input/output signals are passed as separate arguments to the function.

- **Inputs**—Each input signal is passed by value to the procedure. Arguments are passed in pin order.
- **Outputs**—Each output is passed by reference. Arguments are passed in pin order.
- **Info**—The same as described for **Info** in the *Structure-Based Interface* section.

Phases and Error Handling

The phases and error handling within a Standard Procedure is equivalent to the implementation within a subsystem.

Referenced Percent Variables

There exists a capability in SystemBuild, called partitioning of parameter (%var) data. Partitioning allows the same SuperBlock to be customized by using a different Xmath partition containing different values of the parameter data. Partitioning is used exclusively with SuperBlocks, so normally AutoCode is unaffected. However, partitioning does apply to procedures, and special handling must be done within a procedure. For example, consider a procedure named `foo` that has %var parameters named `GAIN` for a gain block, as shown in Example 5-1. In the topmost SuperBlock, there are two references to procedure `foo`, and each reference specifies a different partition, say A and B. Therefore, when procedure `foo` executes from the first reference, the internal gain parameter, `GAIN`, is taken from the A partition. Likewise, when the second reference executes, the parameter `GAIN` is taken from the B partition.

AutoCode supports this capability by:

- Creating separate variables for the %vars from each partition.
- Having generated code in the procedure indirectly access the parameter data.
- Having the procedure's **Info**-struct include %var references.
- Having the caller initialize the **Info**-structure with reference to the appropriate %var.

Note If you specify a specific partition with the %var name in the block form (that is, `A.GAIN`), that %var is directly used, not indirectly referenced.

Example 5-1 Relevant Code to Support Partitioned %vars Within a Procedure

```
/* Xmath variable definitions. */
VAR_FLOAT A_GAIN;
VAR_FLOAT B_GAIN;

/******** Procedures' declarations ********/

/******* Procedure: foo *******/
/***** Inputs type declaration. *****/
struct _foo_u {
   RT_FLOAT foo_1;
};

/***** Outputs type declaration. *****/
struct _foo_y {
   RT_FLOAT foo_2_1;
};

/***** Info type declaration. *****/
struct _foo_info {
   RT_INTEGER iinfo[5];
   VAR_FLOAT *GAIN;
};

/******* Procedure: foo *******/
void foo(U, Y, I)
   struct _foo_u *U;
   struct _foo_y *Y;
   struct _foo_info *I;
{
      RT_INTEGER *iinfo = &I->iinfo[0];

      /***** Output Update. *****/
      /* ---------------------------  Gain Block */
      /* {foo..2} */
      Y->foo_2_1 = (*I->GAIN)*U->foo_1;

      iinfo[1] = 0;
```

```
EXEC_ERROR: return;
}

/******* Subsystem 1  *******/

void subsys_1(U, Y)
   struct _Sys_ExtIn *U;
   struct _Subsys_1_out *Y;
{
      static RT_INTEGER iinfo[4];

      /***** Local Block Outputs. *****/
      RT_FLOAT foo_2_1;
      static struct _foo_u foo_2_u;
      static struct _foo_y foo_2_y;
      static struct _foo_info foo_2_i;
      static struct _foo_u foo_12_u;
      static struct _foo_y foo_12_y;
      static struct _foo_info foo_12_i;
      struct _foo_info *foo_i;

      /******* Initialization. *******/
      if (SUBSYS_PREINIT[1]) {
         iinfo[0] = 0;
         iinfo[1] = 1;
         iinfo[2] = 1;
         iinfo[3] = 1;
         foo_2_i.iinfo[0] = iinfo[0];
         foo_2_i.iinfo[3] = iinfo[3];
         /* Copy in variable pointers of procedure foo */
         foo_2_i.GAIN = &A_GAIN;

         foo_12_i.iinfo[0] = iinfo[0];
         foo_12_i.iinfo[3] = iinfo[3];
         /* Copy in variable pointers of procedure foo */
         foo_12_i.GAIN = &B_GAIN;

         SUBSYS_PREINIT[1] = FALSE;
         return;
      }

      /***** Output Update. *****/
      /* ---------------------------  Procedure SuperBlock */
      /* {foo.2} */
```

```
      foo_2_u.foo_1 = U->t_1_1;
      foo(&foo_2_u, &foo_2_y, &foo_2_i);
      foo_2_1 = foo_2_y.foo_2_1;
      iinfo[0] = foo_2_i.iinfo[0];
      if( iinfo[0] != 0 ) {
         foo_2_i.iinfo[0] = 0; goto EXEC_ERROR;
      }
      /* ---------------------------  Procedure SuperBlock */
      /* {foo.12} */
      foo_12_u.foo_1 = foo_2_1;
      foo(&foo_12_u, &foo_12_y, &foo_12_i);
      Y->foo_12_1 = foo_12_y.foo_2_1;
      iinfo[0] = foo_12_i.iinfo[0];
      if( iinfo[0] != 0 ) {
         foo_12_i.iinfo[0] = 0; goto EXEC_ERROR;
      }
      if(iinfo[1]) {
         SUBSYS_INIT[1] = FALSE;
         iinfo[1] = 0;
      }
      return;

EXEC_ERROR: ERROR_FLAG[1] = iinfo[0];
      iinfo[0]=0;
}
```

Procedure Arguments

All or some of the following arguments need to be passed for each call to the procedure in the following order.

U, Y, S, and I

These arguments are pointers to structures reflecting the procedure's inputs, outputs, states (including nested procedure states) and informational data (including nested procedure informational data) used for communication with the user application or simulation engine.

The inputs to the procedure are provided by the argument U, a pointer to a structure named *_procedure name*_u. This structure is composed of mixed data-typed variables reflecting each procedure input signal and type.

The outputs to the procedure are provided by the argument Y, a pointer to a structure named *_procedure name*_y. This structure is composed of

mixed data-typed variables reflecting each procedure output signal and type.

The states of the procedure are provided by the argument S, a pointer to a structure named *_procedure name*_s. This structure contains the double-buffered private states used in the procedure, a flag to toggle the private states from one buffer to another, and the states structures of all procedures nested in the procedure. The private states are provided by the element *procedure name*_ps, a two-element array of a structure named *_procedure name*_ps. The private states structure is composed of mixed (usually RT_FLOAT) data-typed variables reflecting each state needed in the procedure. The flag to toggle private states is a variable of type RT_INTEGER named *_procedure name*_x whose value is toggled between 0 and 1 by the subsystem invoking the procedure.

The informational data of the procedure is provided by the argument I, a pointer to a structure named *_procedure name*_info. This structure contains:

- Status and control flags stored in an array named iinfo.
- Time-related information stored in an array named rinfo.
- Block parameter data used by block algorithms in the procedure stored in arrays named IP or RP.
- Xmath and Variable block variables used in the procedure stored as pointers to global variables and named after the Xmath variables or Variable block variables.

In addition, information data of the form described for procedures nested in the procedure is provided in the *_procedure name*_i structure as a pointer to a structure named *_nested_procedure name*_info. Table 5-1 and Table 5-2 summarize the elements in structure *_procedure name*_info. These elements, which are used by the procedure, need to be set by the function invoking the procedure.

Table 5-1. Description of Element iinfo in Structure *_procedure name_*info

Array Element	Description
`iinfo[0]`	Error flag.
`iinfo[1]=1`	INIT mode. Initialize.
`iinfo[2]=1`	STATE mode. Compute state derivatives.
`iinfo[3]=1`	OUTPUT mode. Compute outputs in Y.
Note: `iinfo` is an array of `RT_INTEGER` containing status and control flags.	

Table 5-2. Description of Element rinfo in Structure *_procedure name_*info

Array Element	Description
`rinfo[0]`	Current time in seconds.
`rinfo[1]`	Sample interval in seconds (1.0 if procedure is invoked from triggered task).
`rinfo[2]`	Initial time skew in seconds (0.0 if procedure is invoked from triggered task).
`rinfo[3]`	Timing requirement, in seconds, if procedure invoked from triggered task (0.0 if procedure invoked from discrete task).
`rinfo[4]`	Start time in seconds.
Note: `rinfo` is an array of `RT_FLOAT` containing time-related information.	

The parameter arrays `RP` (real parameters) and `IP` (integer parameters) in structure *`_procedure name_`*`i` are used for storing parameter values used by algorithms of blocks in the procedure. During initialization (that is, INIT mode is 1), the procedure initializes the `IP` and `RP` arrays in the structure with the necessary values entered in the model.

If Xmath variables are used as %variables in the model as a value in a block dialog form, or if Variable block variables are used in the model, the variables used in the procedure are passed through the *`_procedure name_`*`info` structure as pointers to global variables named after the Xmath variables used. You need to initialize these pointers.

Extended Procedure Information Structure

The `-epi` option specifies that additional elements to all standard procedure SuperBlock's `info` structure are to be generated. Currently, only one additional element is generated. It is named `caller_id` and is of type `RT_INTEGER`. Example 5-2 and Example 5-3 show an `info` structure declaration (in C) with and without the `-epi` option. Equivalent structures are declared when generating Ada code.

Example 5-2 Regular Procedure info Structure Declaration

```
struct _proc1_info {
   RT_INTEGER iinfo[5];
   RT_FLOAT rinfo[5];
};
```

Example 5-3 Extended Procedure info Structure Declaration (-epi)

```
struct _proc1_info {
   RT_INTEGER iinfo[5];
   RT_FLOAT rinfo[5];
   RT_INTEGER caller_id;
};
```

Caller Identification

The purpose of the `caller_id` element is to provide the unique identifier of the caller, that is, the number of the subsystem task, background, startup, or interrupt procedure SuperBlock. As implied in Example 5-2 and Example 5-3, all subsystem tasks and nonstandard procedure SuperBlocks are assigned a unique identifier. Starting with 1, the tasks and procedures are numbered in this order: subsystem tasks, startup procedures, background procedures, and interrupt procedures.

Startup procedures are unlike the other procedures in that they are called only once during the startup phase. Consequently, it has been shown that easy identification of when a startup SuperBlock is referenced in the `caller_id` element simplifies template programming and code generation. Therefore, startup SuperBlocks are represented as the *negative* of the ordering number. For example, assume a model has three subsystem tasks, three startup procedure SuperBlocks, two background procedure SuperBlocks, and one interrupt procedure SuperBlock. The ordering is shown in Table 5-1.

Table 5-3. Procedure Ordering

Task/Procedure	Unique Identifier
subsystem task 1	1
subsystem task 2	2
subsystem task 3	3
startup procedure 1	–4
startup procedure 2	–5
startup procedure 3	–6
background procedure 1	7
background procedure 2	8
interrupt procedure 1	9

Notice the relative numbering within a task/procedure type. Also, be aware that standard procedures are not given a unique identifier for the purposes of the `caller_id` element. A standard procedure SuperBlock uses the id of its caller as its own when the caller's id is needed.

In addition to declaring the extra element in the `info` structure, the `-epi` option causes AutoCode to assign the unique task/procedure identifier to the `caller_id` element and to use the `caller_id` as an argument for variable block callouts for variable block accesses within a standard procedure SuperBlock. Refer to the *Global Variable Block Callouts* section for more information about variable block callouts.

Compatibility Issues

The use of `-epi` affects all generated procedures. It is not possible to specify some procedures with and some other procedures without the `caller_id` element. Also, subsystem code generated assumes the existence of the `caller_id` element in all standard procedure SuperBlock `info` structures and generates code based on that assumption.

You cannot automatically mix procedures generated with the `-epi` option and procedures generated without the `-epi` option; you must manually add the `caller_id` element to the old procedure's `info` structure declaration. The old procedure will have been generated without the existence of the new element and thus its presence in the structure will not affect the previously generated code as that code never references it. Its effect,

however, is to create additional space when declaring a variable of the `info` structure's type and for the new code generated with the `-epi` option, which assumes the field exists in all procedure `info` structures. However, the old procedure cannot call a procedure generated with the extended field in the `info` structure. Also, if attempting to mix procedures with and without the extended structure, it is not possible to generate valid code if the `-vbco` option is used in conjunction with `-epi`.

Macro Procedure

Macro Procedure SuperBlocks are intended to provide an in-line code generation capability like C-macros provide. However, AutoCode does not generate the definition of the macro procedure. AutoCode only generates a call to the macro, assuming that the macro is implemented elsewhere. Of course, the name generated for the Macro Procedure can be the name of the function rather than a macro, as would always be the case for Ada.

Interface

The interface generated to a Macro Procedure is very similar to the unrolled Standard procedure interface in that inputs and outputs are actual arguments to the procedure.

- **Macro Name**—The name of macro is generated exactly as entered in the macro string in the block form.
- **Additional Arguments**—The macro string can contain additional arguments as specified in the macro string. These arguments are generated before the explicit inputs and outputs of the block.
- **Input**—Each of the inputs to the Macro Procedure are listed in pin order.
- **Output**—Each of the outputs of the Macro Procedure are listed following the inputs in pin order. Notice that outputs are *not* passed by reference.

Example 5-4 is the generated code for a Macro Procedure. Assume that the name of the macro is `printf`; there are two inputs and an optional argument.

Example 5-4 Sample call generated for a Macro Procedure

```
/* ----------------------------  User Macro Block */
/* {mac.22} */
printf("%d %d\n", U->data_1, U->data_2);
```

Asynchronous Procedures

Asynchronous Procedures are procedures that are not regularly scheduled to be executed or directly called from a subsystem or Standard Procedure. In other words, these procedures require some entity outside of the scope of the SystemBuild diagram to invoke them. The following rules apply to asynchronous procedures, but not necessarily to asynchronous subsystems. For details about asynchronous subsystems, refer to the *AutoCode User Guide*.

These procedures share the following characteristics:

- There are no external inputs and no external outputs.
- Dynamic blocks, that is, blocks with explicit states, are not supported.
- Global variable blocks are the only way to communicate between these procedures and subsystems.

Interrupt

Some external interrupt event causes an Interrupt Service Routine (ISR) to call the Interrupt Procedure. You must write all of the mechanisms for your particular target to achieve this behavior.

Background

This code executes when there is no other activity happening in your system. Obviously, this implies some kind of policy for the scheduler to decide when the background procedure should execute. You are required to implement all of the necessary scheduler mechanisms or use an RTOS. The scheduler within the standard C and Ada templates executes all of the background procedures after all of the subsystems execute for each scheduler minor-cycle.

Startup

This type of procedure performs special initializations. The standard template-generated code that executes all of the startup procedures before time = 0.0 but after all of the subsystems have executed their PREINIT phase.

Changing %var Values During Startup

A special feature of the Startup allows the value of a %var to be set at run-time through a Global Variable Block that has the same name as the %var.

Condition Block

SystemBuild provides three variations of the Condition Block: Default, No-Default, and Sequential.

Note AutoCode only supports Standard and Macro Procedures within a Condition Block. AutoCode does *not* support inline procedures within a Condition Block.

Default Mode

This mode requires that at least one procedure within the Condition Block will execute. If there are *n* procedures, there are only *n* – 1 conditions and thus the *n*th procedure will execute if none of the other procedures execute.

No-Default Mode

This mode does not require that at least one procedure will execute. Rather, if all of the conditions fail, the results from the last time-cycle are used. AutoCode stores all of the outputs of the last executed procedure within `R_P` to provide this functionality. Type conversions are performed for non-float data types.

Sequential Mode

This mode independently tests the condition for each of the procedures. Therefore, all, some, or none of the procedures might execute. If none of the procedures execute, the outputs are taken from the last time the procedures executed. Like the no-default mode, the outputs are cached within the `R_P` and type conversions are performed for non-float data.

BlockScript Block

The BlockScript block lets you create a custom algorithm within the context of the block in the diagram. This block uses a scripting language called BlockScript, which AutoCode can translate into C or Ada code. BlockScript provides a generalized programming capability for defining SystemBuild blocks for simulation and code generation.

The BlockScript block allows you to specify conditions and actions, define block inputs, outputs, and parameters, and specify their data types and dimensions. BlockScript then writes the update equations that process the inputs and parameters to produce the outputs. BlockScript I/O can be read by the Data Dictionary. The BlockScript block is designed for general use; however, there are special semantics that apply to the translation of the BlockScript into code that can cause unexpected results if not fully understood.

Inputs and Outputs

The inputs and outputs of a BlockScript block are represented by a set of scalars or vectors. If a vector, the vector might be a fixed size, or a size related to the number of inputs or outputs of the block.

Note The size of the I/O variables and the number of I/O signals of the BlockScript block are recursively related. If you change one, the other also changes.

Inputs and outputs are declared in two steps:

1. Create the names of the variables to be used as inputs and outputs. The order of this signature is critical as it provides a mapping to the input/output pins of the block.
2. Specify the data type and size of the variables.

If you do not specify a data type declaration, BlockScript assumes the variable is a scalar float. The order within the data type declaration is not important.

Example 5-5 shows declarations of inputs and outputs.

Example 5-5 Example BlockScript Block Input/output Declarations

```
inputs : (alpha, beta, gamma);
outputs: result;

float alpha, gamma(5);
integer beta(5), result(10);
```

```
inputs : data;
outputs: control;

float data(:), control(data.size);
```

In Example 5-5, alpha, beta, and gamma are the variables to be used as representations for the inputs. Alpha is a scalar representing the input from pin 1. Beta is an array of integers representing inputs pins 2 through 6. Gamma follows as an array of floats representing pins 7 through 10. The result is just an array of integers representing all of the outputs pins. Because the sizes are explicit, the BlockScript block with that declaration can have only 11 inputs and 10 outputs.

In the second set of declarations in the example, data and control are the input and output variables. The size of data is declared with the colon, which indicates that the size of the variable is the size specified by its classification. Since data is an input, then data will be the size of the number of inputs of the block. The size of control uses a feature of BlockScript that allows for convenient access to attributes of a variable. In this case, the size of the variable is used to declare the size of control. Therefore, control will have the same size as data; hence, the BlockScript block will have the same number of input signals as output signals.

Environment Variables

BlockScript provides a set of environment variables that represent read-only values. Some of these values represent the phases of the subsystem, or some represent data, such as the current time or tolerances. When translated into code, these variables appear as constant variables.

Local Variables

A local variable is a variable that is used exclusively within the block. The value of the local variable is not persistent between invocations of the block. In other words, if the variable is not an input, output, parameter, or environment variable, the variable is local.

Local variables should have an explicit type declaration. However, if you do not have an explicit type declaration, AutoCode creates an implicit declaration of the variable based on its first use within the block, inferring the data type from that expression. This does not include the first use within dead code.

Implicit local variable declarations are allowed because of compatibility with older versions. NI suggests that you always declare your local variables to eliminate possible data type conflicts.

Note If you intend to generate code with the Typecheck feature disabled (refer to the *Selection of a Signal Name* section), you should only use the float data type. If not, there may be type conflicts when generating code.

Init, Output, and State Phases

A subsystem has phases because the blocks within the subsystem need phases of computation. The three phases are intended to be used in a consistent way, just like the standard blocks. There are environment variables that indicate if a particular phase is active. Therefore, the canonical usage is that the appropriate environment variable is used as the guard in an IF statement. This clearly indicates what code is associated with each phase. The intended usages are listed below.

- **Init**—The intent is to initialize any local, output, state, or parameter variable.
- **Output**—The intent is to only compute the outputs of this block as a function of some combination of inputs, parameters, or current state.
- **State**—The intent is to only compute the next state (or state derivative) as a function of some combination of current state, inputs, outputs, or parameters.

Caution No validation is performed to ensure that you write code that conforms to the intended usage of the phases.

In a discrete subsystem, the phases of a block are set in the following way. The first time the block is executed, the Init, Output, and State phases are all active at the same time. For subsequent executions, only Output and State phases are active.

Note Regardless of the order within the script, code for the Output phase always will execute before code for the State phase.

If the BlockScript block has states, place the Init phase within the Output phase to prevent a repeat of the initialization during the State phase.

In a continuous subsystem, the Init and Output phases are active for the first execution. The block can then be executed repeatedly for the State phase, as the integrator integrates the state data for all blocks. On subsequent time points, the cycle repeats except the Init phase is no longer active.

Caution Be careful about any side-effects to persistent data within the Init or Output phases. This can cause mismatches between SystemBuild Simulator and the AutoCode simulation, because the SystemBuild Simulator might execute the block more than once at time = 0.0 for the Init/Output phase, whereas the AutoCode simulation only executes the time = 0.0 Init/Output phase once. Refer to the INITMODE option for the SystemBuild Simulator for more information about its initialization behavior.

Default Phase

If you do not specify a phase and/or all code is not contained within an IF statement guarded by a phase environment variable, that code is generated in the Output phase and, if there is a State phase, that code also is generated in the State phase. Example 5-6 shows the phases.

Example 5-6 Example BlockScript Block Phases

```
Inputs : u;
Outputs: y;
Environment: (INIT, OUTPUT, STATE);
Parameters: wobble;
States: x;
Next_States: xnext;
float u(10), y(10), wobble(10), x(10), xnext(x.size);
integer i;

if OUTPUT then
  if INIT then
    for i = 1:10 do
      wobble(i) = 0.1*i;
      x(i) = wobble(i) * 3.14;
    endfor;
  endif;

  for i = 1:10 do
    y(i) = u(i) * x(i) / wobble(i);
  endfor;
endif;

if STATE then
  for i = 1:10 do
    xnext(i) = x(i) + (x(i) / wobble(i));
  endfor;
endif;
```

States

States within a BlockScript block must conform to special semantics because the subsystem will assume the BlockScript block uses the states as all of the standard blocks do. State semantics within a discrete subsystem are different from those of a continuous subsystem. Therefore, it is possible that a BlockScript block used in a discrete subsystem will not produce correct results in a continuous system.

Local Variables and Phases

A local variable *cannot* be used to pass data between phases, because the different phases occur at different locations in the execution order of the whole subsystem or procedure. That is, local variables *can* be reused for other BlockScript block.

Only inputs, outputs, parameters, and states are guaranteed to be correct between phases. This is shown in Example 5-7 where, within the STATE update, the output is used to pass data between the phases.

Discrete Semantics

SystemBuild has two types of state categories for a BlockScript block in a discrete subsystem: States and Next States. You can specify more than one variable and different data types for the state variables, as with inputs and outputs. The State variable(s) are intended to represent state data from the previous time point. The State variable(s) should only be used for read-only purposes. The Next State variables are intended to represent the state data to be used in the next time point. The Next States variables can be both read and written. The very last thing that a subsystem does is to swap the data from the Next State variables into State variables for the next time point. You can view state data as a way to provide double-buffered persistent data.

Example 5-7 shows how to keep a running total of the input values.

Example 5-7 Discrete BlockScript Block Example (Keeping a Running Total)

```
Inputs: u;
Outputs: y;
Environment: (INIT, OUTPUT, STATE);
States: current_total;
Next_States: new_total;
float u,y,current_total, new_total;

if OUTPUT then
  if INIT then
    current_total = 0.0;
  endif;

  y = u + current_total;
endif;

if STATE then
```

```
  new_total = y;
endif;
```

Example 5-8 Generated Code from Example 5-7

```
void subsys_1(U, Y)
   struct _Sys_ExtIn *U;
   struct _Subsys_1_out *Y;
{
   /***** States Array. *****/
   static struct _Subsys_1_states ss_1_states[2] = {{0.0}, {0.0}};

   /***** Current and Next States Pointers. *****/
   static struct _Subsys_1_states *X = &ss_1_states[0];
   static struct _Subsys_1_states *XD = &ss_1_states[1];
   static struct _Subsys_1_states *XTMP;

   /***** Output Update. *****/
   /* ----------------------------  BlockScript Block */
   /* {bsb..2} */
   if (INIT) {
      X->bsb_2_S1 = 0.0;
   }

   Y->bsb_2_1 = U->bsb_1 + X->bsb_2_S1;

   /***** State Update. *****/
   /* ----------------------------  BlockScript Block *
   /* {bsb..2} */
   XD->bsb_2_S1 = Y->bsb_2_1;

   /***** Swap state pointers. *****/

   XTMP = X;
   X = XD;
   XD = XTMP;}
```

Continuous Semantics

The state data within a continuous subsystem are called States and State Derivatives. These look very similar to the discrete equivalents, except that State data is integrated by the integrator algorithm. This is what is intended by using states within the continuous system. Therefore, if you just translated the previous discrete BlockScript block example into using States and State Derivatives, the results would be quite different. So, States are more than just persistent data in the continuous subsystem.

Looping Concepts

BlockScript contains features within the language to implement loops. However, loops have a far-reaching effect on the generated code. This section describes the effects and limitations of loops in the generated code.

Terminology

The discussion of loops in BlockScript cannot proceed until some terms are defined. We must introduce the concept of *rolled* and *unrolled* loops. A rolled loop is just a loop in the standard programming sense. An unrolled loop is not a loop at all. An unrolled loop is the body of the loop generated for each iteration of the loop. In other words, an unrolled loop is the expansion of that loop over the range of the loop bounds.

Other terms include *soft-subscript* and *hard-subscript*. A soft-subscript is an array subscript that is run-time bound to another variable. A hard-subscript is an array subscript that is known during code generation, that is, a constant value.

Loops and Scalar Code

When AutoCode is used for scalar code generation, code for the BlockScript must be generated using scalar variables. However, the syntax of BlockScript uses arrays, especially when referring to the inputs and outputs of a block. So, when generating scalar code, AutoCode must translate array variables in BlockScript into a set of scalars. This transformation causes a significant increase in the size of code as well as prevents some types of algorithms from being directly implemented. The scalar representation of BlockScript variables applies to inputs, outputs, and states. Parameters and local variables of the BlockScript can be either scalar or arrays. This means that only hard-subscripts can be used with inputs, outputs, and states for scalar code generation.

In Example 5-9, a hard-subscript, `i`, is used to access both inputs and outputs. The reason `i` is a hard-subscript is that the range of `i`—that is, the values of 1 through to the number of inputs (`u.size`)—is known when the code is generated. AutoCode then can unroll the loop so as to translate `y(i)` into a single scalar variable for each value of `i`. The same is true for `u(i)`.

Example 5-9 Hard-Subscript Used to Access an Array

```
Inputs:  u;
Outputs: y;
float u(:), y(u.size);
integer i;

for i = 1:u.size do
  y(i) = u(i) * 2*i;
endfor;
```

In Example 5-10, a soft-subscript, `j`, is being used to access both inputs and outputs. That script will not generate code. The reason `j` is a soft-subscript is that the range of `j`, that is, the values of 1 through to the value of the first input `u(1)`, is not known when code is generated because the upper-bound of `j`'s range is the value of the first input. Because the range of `j` is not known, AutoCode cannot unroll the loop so as to translate the input/output arrays into their scalar representation.

Example 5-10 Soft-Subscript Example

```
Inputs:  u;
Outputs: y;
integer u(:), y(u.size);
integer j;

for j = 1:u(1)do
  y(j) = u(j) + j;
endfor;
```

Rolling Loops with Scalar Code Generation

Although inputs, outputs, and states are translated into scalars, parameters and local variables of the BlockScript can be either scalar or array variables in the generated code. Thus, a common trick has been to copy the inputs and outputs into local variables that are arrays and use those variables for the algorithm. Refer to Example 5-11.

Example 5-11 Local Variables Used to Allow Loops in Scalar Code Generation

```
Inputs:  u;
Outputs: y;
float u(:), y(u.size), local_u(u.size), local_y(y.size);
integer i,j;

for i = 1:u.size do
  local_u(i) = u(i);
endfor;

for i = 1:u.size do
   for j = i:u.size do
     local_y(i) = local_u(i) + local_u(j);
   endfor;
endfor;

for i = 1:y.size do
  y(i) = local_y(i);
endfor;
```

Although the generated code is not very efficient, the amount of code that is generated for Example 5-11 is far less than if the local variables were not used for the same algorithm.

Vectorized Code

AutoCode can generate vectorized code (refer to Chapter 6, *Vectorized Code Generation*) and at the most basic level, it means that arrays will be used instead of scalar variables. Given that, AutoCode does not have to translate the BlockScript inputs, outputs, and states into scalars, rather it will generate arrays. As a result, when AutoCode is generating vectorized code, the soft-script limitation does not apply. Therefore, if generating vectorized code, Example 5-10 generates code, and the trick to use local variable as shown in Example 5-11 is not needed.

Types of Loops

BlockScript provides two different types of loops. Each loop type has a specific usage that effects what you are allowed to use within the loop body.

- **WHILE** Loop—Always generate a rolled loop for both scalar and vectorized code. A soft-subscript is never allowed to access inputs, outputs, or states.

- **FOR** Loop—Can generate either a rolled or unrolled loop depending upon the range of the loop subscript and whether or not scalar code is generated.

Table 5-4. Scalar Code Semantics for the Loop Types

Loop Type	Soft-Subscript	Rolled Loop
WHILE	Not for inputs, outputs, or states	Always
FOR	Not for inputs, outputs, or states	No, unless the bounds of the loop are known and only local variables are used in the loop body

Table 5-5. Vectorized Code Semantics for the Loop Types

Loop Type	Soft-Subscript	Rolled Loop
WHILE	Not for inputs, outputs, or states	Always
FOR	Yes	Yes

Examples of Rolled and Unrolled Loops

Example 5-12 Unrolled Loop from Example 5-9

```
/***** Output Update. *****/
/* ----------------------------  BlockScript Block */
/* {bsb..2} */
bsb_1 = indata_1 * 2;
bsb_2 = indata_2 * 4;
bsb_3 = indata_3 * 6;
bsb_4 = indata_4 * 8;
bsb_5 = indata_5 * 10;
```

Example 5-13 Rolled Loop from Example 5-10, Using Vectorized Code

```
/***** Output Update. *****/
/* ----------------------------  BlockScript Block */
/* {bsb..4} */
for(j=1; j <= indata[0]; j++) {
  bsb[-1+j] = indata[-1+j] + j;
```

Parameters

Parameters represent data that can be used to provide data to tune the algorithm by representing coefficients in equations or persistent data somewhat like states. Parameters are implemented as persistent data when generated into code. Parameters can be initialized with hard-coded values or initialized using a %var as specified from the block dialog within the SuperBlock Editor.

Caution Parameters are designed for read-only data. Do not update a parameter in a continuous model. It is impossible to predict the number of times a block will be executed in a continuous system due to multiple calls of the Init, Output, or State phases. Therefore, it is not known when to update the parameter value. For a continuous system, use the states mechanism or use an input and output of the block connected to a Variable Block to provide persistent data.

Using Parameters Instead of States in a Discrete Model

Although parameters are intended to be read-only data, it is possible within a discrete model to update a parameter because the execution of a block is known, that is, a block will be called only once for the Init phase, once for the Output phase at each time point, and once for the State phase (if it has states) at each time point.

The benefits of using parameters for persistent data instead of states are purely code generation related, specifically, reduced amount of code and reduced amount of data by over 50% as compared to using states. However, you must change the way the algorithm is coded to properly handle the parameter update.

Note National Instruments does not recommend that you replace all uses of states with parameters. Rather, this section points out a special-case method to code a BlockScript block to produce more efficient code for simple cases of persistent data.

The proper updating of a parameter value is a matter of guarding when the update of the parameter occurs. For most cases, you will only want to update the parameter in the Output phase. Thus, the BlockScript code reflects this as in Example 5-14.

Example 5-14 BlockScript Block Example with Updating of a Parameter

```
Inputs: u;
Outputs: y;
Environment: (OUTPUT, INIT);
Parameters: total;
float u,y,total;

if OUTPUT then
  if !INIT then
    total = total + u;
  else
    total = u;
  endif;
  y = total;
endif;
```

Notice that you also must prevent an update during the Init phase, which requires the use of the nested IF statement. Also, the nested IF statement is used to ensure that the SystemBuild Simulator updates the parameter only during the Output phase and not the Init phase.

Caution Failure to properly guard the update of the parameter data will cause the parameter to update more frequently than intended.

Example 5-15 Generated Code for Example 5-14

```
void subsys_1(U, Y)
  struct _Sys_ExtIn *U;
  struct _Subsys_1_out *Y;
{
  static RT_FLOAT total = 0.0;
  static RT_INTEGER INIT;

  if (SUBSYS_PREINIT[1]) {
       INIT = 1;
       return;
  }

  /***** Output Update. *****/
  /* ----------------------------  BlockScript Block */
  /* (bsb..22) */
  if (!INIT) {
    total = total + U->bsb_1;
  }
```

```
else {
  total = U->bsb_1;
}
Y->bsb_22_1 = total;
INIT = 0;
```

Optimizations

When translating a BlockScript block into source code, certain optimizations are automatically done. These optimization can reduce direct traceability from the script to the code at the expense of tighter code.

Constant Propagation/Hard-Coding

A local variable that is assigned a constant value is replaced with its value when code generates. The name of that constant local variable does not appear in the generated code. If it is important to see the symbolic name in the generated code, consider using a parameter instead and `-p` option for code generation.

Dead Code Elimination

Code that is guaranteed to never execute is called dead code. When dead code is detected (as shown in Example 5-16), that code is not translated into generated source code. You can use this to your advantage by writing the script in a more general way, taking advantage of special cases when they occur while not generating dead code. Dead code is most commonly used by combining constants and `if` statements. If the guard of the `if` statement is a constant, then only one of the two branches is translated into generated code.

Example 5-16 BlockScript Block Code with Dead Code

```
Inputs: u;
Outputs: y;
float u(y.size),y(:);
float threshold;

threshold = 0.001;

if threshold < 0.005 then
  for i = 1:y.size do
    y(i) = u(i) / threshold;
  endfor;
else
```

```
  for i = 1:y.size do
    y(i) = 0.0;
  endif;
endif;
```

Example 5-17 Generated Code for BlockScript Block Example 5-16

```
/***** Output Update. *****/
/* ---------------------------  BlockScript Block */
/* {deadbsb..2} */
for (i=1; i<=5; i++) {
  deadbsb[-1+i] = U->deadbsb[-1+i]/0.001;
}
```

Notice that in the generated code, only the true-branch of the BlockScript `if` statement is generated, and that the local variable threshold was hard-coded.

Implicit Type Conversion

Compared to Ada and even C, BlockScript is a very loosely typed language. Thus, you can ignore most data type issues and focus on the algorithm. However, when translated to code, there might be excessive data type conversions that severely penalize performance. NI recommends that if you want to maximize the performance of the generated code, you eliminate any implicit conversions in your BlockScript block.

Special Directives

The BlockScript language allows special directives that force certain attributes or conditions to be applied. Table 5-6 describes the currently supported directives.

Table 5-6. BlockScript Special Directives

Directive	Description	Example
`{volatile}`	Forces the declaration of the listed variable(s) in the generated code.	`{volatile} integer data;`
`{unroll}`	Forces the loop to be unrolled instead of rolled. Only effective when vectorizing code or when nesting beyond the 8-level limit.	`{unroll} for i = 1:10 do`

For more information about BlockScript, refer to the *BlockScript User Guide* or the *MATRIXx Help*.

UserCode Block

The purpose of the UserCode Block (UCB) is to interface your existing code with AutoCode-generated source code. A UCB is typically used to access low-level device drivers or any algorithm not easily modeled within the diagram.

Unfortunately, there are various types of UCBs with slightly different interfaces. The following two categories divide the types of UCBs:

- UCBs intended to be linked directly back into the SystemBuild Simulator
- UCBs intended to be linked with AutoCode-generated code

For a discussion of the details of the UCB interface, refer to Chapter 2, *C Language Reference*, and Chapter 3, *Ada Language Reference*.

Phases of the UCB

The UserCode Block has Init, Output, and State update phases. Due to reasons of efficient code generation in a discrete system, more than one phase may be enabled for a call to the UCB. Specifically, a UCB in a discrete system will have both Output and State phases enabled during the same call to the UCB. In a continuous system, the Output and State phases are not enabled during the same call and thus the UCB is called once for the Output phase then again for the State phase.

You can prevent potential problems by coding your UCB such that the code for each of the phases is in the following order: Init, Output, then State.

Indirect Terms

If any of the terms of a UCB are specified as indirect, either in a continuous or a discrete model, the UCB will be called twice within the subsystem, once for just the Output phase and then once for the State phase.

Note It is possible to get warnings about using a variable before it is set when compiling code which contains UCBs with indirect terms. This is because the UCB will be ordered in such a way that the inputs to the indirect terms have not been computed. This is exactly what is meant with indirect terms and this warning will not cause a computational error, assuming the UCB obeys its own indirect term specification.

Parameterized UCB Callout

A UCB can be defined with %var parameterized data for the UCB's real parameters (RP) and integer parameters (IP). When used, AutoCode generates code that passes the %var variable as the actual of the UCB callout. A new option, -ucbparams, creates a temporary array within the subsystem code and passes the temporary arrays as actuals of the UCB callout. The temporary arrays are initialized with the values of the %var when the code was generated. Using the temporary arrays allows the UCB to change its RP/IP parameters without affecting the %var.

Example 5-18 clarifies this. Assume a UCB is parameterized with the %var named floatdata for the UCB's real parameters and the %var named integerdata for the UCB's integer parameters.

Example 5-18 Relevant Code for UCB Call

```
/*--------------------------------------*/
/* USRxx(INFO,T,U,NU,X,XDOT,NX,Y,NY,RP,IP) */
   usr01(&I->usr01_13_i, TIME, &usr01_13_u[0], 1, &dummp_f[0],
         &dummy_f[0], 0, &usr01_13_y[0], 1, &floatdata[0],
         &integerdata[0]);
/*--------------------------------------*/
Relevant code for UCB call with -ucbparams option
/*------------------------------------------------------------*/
   static RT_INTEGER I_P[4];
   static RT_FLOAT R_P[3];
   static const RT_INTEGER _I_P[4] = {3, 5, 30, 40};
   static const RT_FLOAT _R_P[3] = {2.05, 3.45, 7.63};
   if (SUBSYS_PREINIT[1]) {
      for( cnt=0;cnt<4;cnt++) {
        I_P[cnt] = _I_P[cnt];
      }
      for( cnt=0;cnt<3;cnt++) {
        R_P[cnt] = _R_P[cnt];
      }
   }
 /* USRxx(INFO,T,U,NU,X,XDOT,NX,Y,NY,RP,IP)*/
 usr01(&I->usr01_13_i, TIME, &usr01_13_u[0], 1, &dummp_f[0], &dummy_f[0],
         0, &usr01_13_y[0], 1, &R_P[0], &I_P[0]);
/*------------------------------------------------------------*/
```

Software Constructs

These blocks provide the software constructs that are typically found in any imperative programming language such as C, Pascal, and FORTRAN. The following software constructs are supported in SystemBuild and AutoCode:

- IfThenElse Blocks (for conditional execution)
- WHILE Blocks (for iterative computations)
- Sequencers (for sequencing purpose)
- Local Variable Blocks (for storing temporary data)

For the highest level of flexibility, the designs of the software construct block require you to handle all of the details in implementing your algorithm.

IfThenElse Block

The IfThenElse Block has two sections: one section contains the condition to be evaluated and the other contains blocks that need to be executed if the condition evaluates to True. An ELSE-IF or ELSE Block can be associated with an IfThenElse Block. An ELSE-IF Block has a condition section while an ELSE Block does not have one.

An IfThenElse Block in the block diagram along with the associated ELSE-IF and ELSE Blocks is generated as an `if-else`, `if-else` compound statement in C and as an `if-elsif-else` statement in Ada. For every if and else, if statement, the corresponding condition is generated, followed by a branch that contains code for the blocks contained in it. Block ordering with a container frame is just like that of a subsystem.

Variable Blocks must be used to get data out of the computations within the IF branches, because there may be different numbers of outputs computed from each branch. It is recommended that host local variable blocks be used for this purpose.

WHILE Block

The WHILE block is a construct that allows a set of blocks to be repeatedly executed during the subsystem's scheduled time-frame during one invocation of a subsystem or a procedure. AutoCode implements this construct as a while statement in C and as a loop statement in Ada.

BREAK Block

The WHILE construct indefinitely iterates unless a BREAK Block is executed within the loop. You are responsible for properly detecting an exit condition and using it as an input to a BREAK Block inside the loop. If the input to the BREAK Block is TRUE, then the loop that is immediately enclosing the BREAK Block is exited. This block is implemented as a break statement in C and as an exit statement in Ada.

CONTINUE Block

A CONTINUE Block provides another way to control the execution of blocks within a loop. By using a CONTINUE Block, the execution can be immediately continued with the next iteration, instead of executing blocks that follow the CONTINUE Block. You are responsible for detecting a continue condition and using it as an input to a CONTINUE Block inside the loop. The CONTINUE Block continues execution with the next iteration if its input evaluates to TRUE. This construct is implemented as a continue statement in C, while a `goto` statement is used in Ada, and the target of the goto is the first statement in the loop.

Like the IfThenElse Block, the requirements for an iteration mechanism are varied, and our design allows for the most flexible algorithm design at the expense of increased diagram complexity to specify all of the details about the sequencing and termination within the loop.

Local Variable Block

Local Variable Blocks are implemented as automatic variables in the generated code. They are strictly temporary and are used to pass data from one computation to another. The Scope of a local variable block is local to that of the enclosing subsystem or procedure. The local variable block is active only when the enclosing procedure or subsystem is invoked.

Thus, Local Variable Blocks cannot be used to communicate data between procedures or subsystems, but could be used to communicate data from one SuperBlock to another if both SuperBlocks are mapped to a single subsystem. Refer to subsystem mapping information described in the *Subsystems* section. Also, the recommended way of passing data from one software construct to another, or from a software construct to the SuperBlock in which it is contained, is through local variables.

The Read from Variable Block optimization also is supported for Local Variable Blocks. Refer to Chapter 7, *Code Optimization*, for more details on optimization. Also, the sequencing of local variable blocks is similar to that of global variable blocks explained in the *Global Variable Blocks* section.

Sequencer Block

These blocks are used in the model to control the execution order of blocks. The sequencer block makes sure that all of the blocks to its left are executed before the blocks to its right are executed. In the generated code there is no representation of sequencer blocks, but their presence forces AutoCode to generate code for the blocks that are to its left first, followed by the blocks that are to its right. Refer to the *Block Ordering* section.

Difference Between Local and Global Variable Blocks

Local and Global variable blocks are identical except for their lifetime and scope.

Scope

Global variable blocks are implemented as global variables and are visible throughout the system (have system scope). Local variables are implemented as automatic variables, which are strictly temporary. The scope of a local variable is local, that is, it is only visible to the procedure (or subsystem) where it is defined and used.

Lifetime

The value of a global variable block persists until the program is terminated. On the other hand, the value of a local variable is not remembered from one invocation of a procedure (or subsystem) to the subsequent invocation. Local Variable blocks have to be initialized properly before they can be used.

Continuous Subsystem

For AutoCode purposes, a continuous subsystem is similar to a discrete subsystem. However, whereas a model can have several discrete subsystems, a model can only contain one continuous subsystem. This limitation is required to prevent unexpected results of the execution order. From a continuous point of view, the continuous subsystem is

representative of a sequence of equations. These equations are sensitive (that is, potentially numerically unstable) to the integration algorithm and order in which the equations are computed. Introducing multiple continuous subsystems or procedures introduces arbitrary boundaries within the equation sorting that can affect the stability of the total system. Another reason for having only a single continuous subsystem is to minimize the complexity of the integration algorithm.

Explicit Phases

As in a discrete subsystem, a continuous subsystem also has an Init, Output, and State phases. However, after comparing the code generated for the different subsystem types, you will see that there are explicit guards protecting each phase within the continuous subsystem. This is required because each of the phases might be called more than one time for a particular time point and independently from each other.

Integrator

A continuous subsystem does not have a period rate to be executed. Rather, we use a continuous sampling rate (CSI) to indicate the frequency of the computation of the continuous subsystem. In a real-time environment, you must make sure the CSI is large enough to account for computation delay. The scheduler uses the CSI to pass control over to an integrator. The integrator is responsible to call the continuous subsystem for each state computation of the integration algorithm.

Limitations

The following is a summary of limitations of the generated code for a continuous subsystem.

- Only fixed-step integrators are supported.
- There is a slight mismatch between SystemBuild Simulation results and AutoCode simulation results. AutoCode does not interpolate the inputs between the explicit time in the time-vector (t) and the time ($t + h$), where h is the integration interval. AutoCode integrators keep the inputs constant at those points.
- States and derivatives within a continuous subsystem are always `RT_FLOAT` data type, which is usually defined within the Stand-Alone Library as double-precision.

- Algebraic loops are not supported.
- AutoCode only performs a single initialization pass at time = 0.0. This corresponds to the SystemBuild Simulation options of `INITMODE=0` or `ACTIMING`.

Multiprocessor Code Generation

Generation for a multiprocessor target is supported by AutoCode by heavily relying on a specialized template to generate a framework for the target. For the most part, generating for multiple processors does not affect the generated code within subsystems. However, the major differences start to appear when handling the data for subsystem interfaces, %vars, Variable Blocks, and asynchronous procedures.

Note A multiprocessor template is not provided in the AutoCode distribution.

Shared Memory Architecture

In a multiprocessor system, subsystems are distributed across difference processors. These subsystems must pass signals between each other and can share common external inputs. Default AutoCode multiprocessor code generation assumes a shared memory architecture and assumes all system and subsystem external inputs and outputs are within a single data structure named `mbuf`. Data included in this structure includes external system input, external system output, data stores, and double-buffered subsystem outputs. Subsystem inputs are handled indirectly because of the double buffering.

Example 5-19 shows the sample and hold phase of subsystem 1.

Example 5-19 Sample and Hold Phase of Subsystem 1

```
subsys_1_in.throttle = ss5_outr->throttle;
subsys_1_in.Brake = mbuf->sys_extin.Brake;
subsys_1_in.PDown = mbuf->sys_extin.PDown;
```

Distributed Memory Architecture

AutoCode also supports a multiprocessor architecture that uses distributed memory instead of shared memory. AutoCode does this by generating callouts (that is, macros) instead of the explicit code, and passes all of the necessary data to the callout. Unfortunately, there are potentially a large number of callouts to support various combinations of functionality and data type of the arguments. Use the `-smco` option to generate those callouts.

Example 5-20 is the same as Example 5-19 except that it has callouts.

Example 5-20 Example with Just the Callouts

```
subsys_1_in.throttle = ss5_outr->throttle;
GET_LOCF_FROM_MBUFF(&subsys_1_in.Brake, &mbuf->sys_extin.Brake);
GET_LOCF_FROM_MBUFF(&subsys_1_in.PDown, &mbuf->sys_extin.PDown);
```

Shared Memory Callouts

The following are the four sets of the callouts (that is, macros) that must be implemented if you generate code with the `-smco` option. The variations of the callouts are to support combinations of float, integer, and Boolean data types.

Callout Naming Convention

The callouts follow a simple naming convention. The convention is a concatenation of operation type, destination, destination data type, source, and source data type. Operation type includes `UPDATE` (write) and `GET` (read). Destination and source are `MBUFF` (shared memory) and `LOC` (local data). Data types are F (float), I (integer), and B (Boolean).

1. Copy data into shared data:

   ```
   UPDATE_MBUFF_WITH_LOCF(dest,src)
   UPDATE_MBUFB_WITH_LOCB(dest,src)
   UPDATE_MBUFI_WITH_LOCI(dest,src)
   UPDATE_MBUFF_WITH_LOCI(dest,src)
   ```

2. Copy between two shared data elements:

   ```
   UPDATE_MBUFF_WITH_MBUFF(dest,src)
   UPDATE_MBUFB_WITH_MBUFB(dest,src)
   UPDATE_MBUFI_WITH_MBUFI(dest,src)
   UPDATE_MBUFF_WITH_MBUFI(dest,src)
   ```

3. Copy a block of local data into shared data:

   ```
   UPDATE_MBUF_WITH_LOCBLK(dest,src,size)
   ```

4. Copy shared data into local data:

   ```
   GET_LOCF_FROM_MBUFF(dest,src)
   GET_LOCB_FROM_MBUFB(dest,src)
   GET_LOCI_FROM_MBUFI(dest,src)
   GET_LOCF_FROM_MBUFI(dest,src)
   ```

Mapping Command Options

There is a set of command options that provide a way to direct AutoCode to generate code for specific functions on a specific processor. This lets you load-balance your system by being able to shift code from one processor to another. The entities that can be mapped to a specific processor include subsystems, background procedures, startup procedures, and interrupt procedures. Refer to the *AutoCode User Guide* for information on how to specify the maps.

Fixed-Point Support for Multiprocessor AutoCode

AutoCode's capability to generate code for multiprocessor hardware has been strengthened with fixed-point data type support. If a multiprocessor target's shared memory architecture prevents direct access to variables (such as alignment problems or distributed memory), AutoCode must generate callouts instead of assignment statements. The callouts are generated when the `-smco` option is used and there are different callouts to deal with different data types. In this section, these fixed point callouts are described in detail. For more details regarding multiprocessor code, refer to the appropriate chapter in the manual.

Definitions and Conventions

The following list presents the terms commonly used when referring to the shared memory callouts.

- `MBUF` refers to shared memory.
- `LOC` stands for local memory.
- `SBYTE` stands for signed byte.
- `UBYTE` stands for unsigned byte.
- `SSHORT` stands for signed short.
- `USHORT` stands for unsigned short.

- SLONG stands for signed long.
- ULONG stands for unsigned long.

The naming convention of the callouts uses the terms listed above and associates from right to left. The following is an example of a callout.

```
UPDATE_MBUFSBYTE_WITH_LOCSBYTE(x, y)
```

The value of the local variable x of type signed byte is assigned to a shared memory variable y of type signed byte. All of these callouts assume that the data type of x and y are identical except when noted.

Shared Memory Fixed-Point Callouts in AutoCode/C

AutoCode/C generates the callout when needed. You must provide the implementation of the callouts. You can choose to use macros or procedure calls and whether or not the implementation is generated from within the template. You can update shared memory as follows.

From Local Memory

```
UPDATE_MBUFSBYTE_WITH_LOCSBYTE(x, y)
UPDATE_MBUFUBYTE_WITH_LOCUBYTE(x, y)
UPDATE_MBUFSSHORT_WITH_LOCSSHORT(x, y)
UPDATE_MBUFUSHORT_WITH_LOCUSHORT(x, y)
UPDATE_MBUFSLONG_WITH_LOCSLONG(x, y)
UPDATE_MBUFULONG_WITH_LOCULONG(x, y)
```

From Shared Memory

```
UPDATE_MBUFSBYTE_WITH_MBUFSBYTE(x, y)
UPDATE_MBUFUBYTE_WITH_MBUFUBYTE(x, y)
UPDATE_MBUFSSHORT_WITH_MBUFSSHORT(x, y)
UPDATE_MBUFUSHORT_WITH_MBUFUSHORT(x, y)
UPDATE_MBUFSLONG_WITH_MBUFSLONG(x, y)
UPDATE_MBUFULONG_WITH_MBUFULONG(x, y)
```

From Shared Memory (mixed data types)

```
UPDATE_MBUFSBYTE_WITH_MBUFF(x, y, convert_macro_name)
UPDATE_MBUFF_WITH_MBUFSBYTE(x, y, convert_macro_name)
UPDATE_MBUFUBYTE_WITH_MBUFF(x, y, convert_macro_name)
UPDATE_MBUFF_WITH_MBUFUBYTE(x, y, convert_macro_name)
UPDATE_MBUFSSHORT_WITH_MBUFF(x, y, convert_macro_name)
UPDATE_MBUFF_WITH_MBUFSSHORT(x, y, convert_macro_name)
UPDATE_MBUFUSHORT_WITH_MBUFF(x, y, convert_macro_name)
```

```
UPDATE_MBUFF_WITH_MBUFUSHORT(x, y, convert_macro_name)
UPDATE_MBUFSLONG_WITH_MBUFF(x, y, convert_macro_name)
UPDATE_MBUFF_WITH_MBUFSLONG(x, y, convert_macro_name)
UPDATE_MBUFULONG_WITH_MBUFF(x, y, convert_macro_name)
UPDATE_MBUFF_WITH_MBUFULONG(x, y, convert_macro_name)
```

The third argument, *convert_macro_name*, is the name of the fixed-point conversion macro that is used for conversion between fixed-point and floating-point numbers.

Reading Shared Memory

These callouts assign the values of the shared variable `y` to the local variable `x`.

```
GET_LOCSBYTE_FROM_MBUFSBYTE(x, y)
GET_LOCUBYTE_FROM_MBUFUBYTE(x, y)
GET_LOCSSHORT_FROM_MBUFSSHORT(x, y)
GET_LOCUSHORT_FROM_MBUFUSHORT(x, y)
GET_LOCSLONG_FROM_MBUFSLONG(x, y)
GET_LOCULONG_FROM_MBUFULONG(x, y)
```

Shared-Memory Fixed-Point Callouts for AutoCode/Ada

The shared memory callouts for fixed-point data types are not supported in this release. However, shared memory fixed-point is supported for Ada code generation as long as the `-smco` option is not used.

Shared Variable Block Support

AutoCode supports shared variable blocks, that is, the same variable block used on more than one processor. AutoCode generates an indirect reference to a shared variable block variable through a pointer. Refer to Example 5-21. This pointer is referenced from an array of pointers. AutoCode always uses the 0th element of the shared variable block pointer array (`isi_varblk`). This is to provide indirection into the shared memory region of the target hardware. The pointer is a pointer to a data structure containing the declarations of the shared variable blocks. An instance of that structure should be declared in the shared memory region of the target hardware and the pointer in `isi_varblk[0]` set to that instance.

Caution It is your responsibility to create the pointers, data structures, and shared memory region for the multiprocessor target hardware. Also, use the variable block callouts to ensure coherency of the shared data.

Example 5-21 shows template code to generate the required structure and pointer. All of the necessary information about the shared variable blocks is accessible from within the template using parameter information.

Example 5-21 Template Code to Generate Required Shared Variable Block Structures (C)

```
@IFF multiprocessor_b@@

@shared_count = 0@@
@
/****  declare shared variable block structure ****/
static struct _shared_varblk {
@LOOPP k=0, k lt nvars_i, k=k plus 1@@
@IFF vars_prsr_scope_li[k] eq 2 or vars_prsr_scope_li[k] eq 3@@
@shared_count = shared_count + 1@@
@
@/declare shared variable block
@
    @vars_typ_pfix_ls[k]@  @vars_ls[k]@
@
@/generate dimensions, if array
@offset = vars_sfix_dim_start_li[k]@@
@LOOPP m=0, m lt vars_typ_sfix_dim_li[k], m=m plus 1@@
[@vars_typ_sfix_li[offset]@]@
@offset = offset plus 1@@
@ENDLOOPP@@
;
@ENDIFF@@
@ENDLOOP@@
@/if no varblks shared, must create a dummy element
@IFF shared_count eq 0@@
    RT_INTEGER ignore;
@ENDIFF@@
};

/**** declare shared variables in shared memory ****/
#pragma SHARED_MEM_BEGIN
   struct _shared_varblk shared_var_blks;
#pragma SHARED_MEM_END
```

```
/**** declare pointer to shared variables ****/
volatile struct _shared_varblk *isi_varblk[1] = {&shared_var_blks};


@ENDIFF@
```

Example 5-21 assumes the existence of a fictional shared memory target such that the compiler supports `#pragmas` to declare the shared memory region. Your compiler and architecture will most likely have a different mechanism.

The following list shows the requirements for shared variable block support. All of the steps can be accomplished within the template using template parameters.

- Create a structure (record) containing the name and data type of the shared variable blocks in the model. This structure should be visible to the code on each processor, or must be declared static. The source file(s) for each processor must have a declaration of the structure.
- Declare an instance of the structure containing the shared variable blocks into shared memory. This will vary, depending on your compiler's and target's shared memory architecture.
- Declare a static variable named `isi_varblk` that is an array (or a pointer in C) and assign the 0th element to the location in shared memory where the shared variable block structure is placed. Refer to Example 5-22.

For Ada, the template code will be identical except that you must transliterate the C syntax into Ada syntax and the shared memory allocation will be different.

Example 5-22 Shared Variable Block Generated Code with Callouts (-vbco)

```
/* ------------------------------ Read from Variable */
Enter_Shared_Varblk_Section(4);
   proc2_4_1 = isi_varblk[0]->block5[0];
   proc2_4_2 = isi_varblk[0]->block5[1];
Leave_Shared_Varblk_Section(4);

/* ------------------------------ Write to Variable */
Enter_Shared_Varblk_Section(4);
   isi_varblk[0]->block5[0] = proc2_4_1;
   isi_varblk[0]->block5[1] = proc2_4_2;
Leave_Shared_Varblk_Section(4);
```

Shared Memory Callout Option

AutoCode supports a shared memory callout for all access to elements in shared memory. Callouts are generated when the `-smco` option is specified. The previous discussion about shared variable blocks still applies. However, the generated code is different, and you must supply the definitions of the callouts.

Read Shared Variable Block Callouts

There are currently four callouts used when reading from a shared variable block. The difference is to accommodate different data types. The prototypes are:

```
RT_FLOAT Read_Shared_Varblk_Float(long offset);
RT_INTEGER Read_Shared_Varblk_32(long offset);
RT_INTEGER Read_Shared_Varblk_16(long offset);
RT_INTEGER Read_Shared_Varblk_8(long offset);
```

There is a callout for `RT_FLOAT`, 32-bit (`RT_INTEGER`, `RT_ULONG`, and `RT_SLONG`), 16-bit (`RT_USHORT` and `RT_SSHORT`) and 8-bit (`RT_UBYTE` and `RT_SBYTE`) data types.

Example 5-23 uses the `Read_Shared_Varblk...` syntax.

Write Shared Variable Block Callouts

There are currently four callouts used when writing to a shared variable block. The difference is to accommodate different data types. The prototypes are:

```
void Write_Shared_Varblk_Float(long offset, RT_FLOAT value);
void Write_Shared_Varblk_32(long offset, RT_INTEGER value);
void Write_Shared_Varblk_16(long offset, RT_INTEGER value);
void Write_Shared_Varblk_8(long offset, RT_INTEGER value);
```

There is a callout for `RT_FLOAT`, 32-bit (`RT_INTEGER`, `RT_ULONG`, and `RT_SLONG`), 16-bit (`RT_USHORT` and `RT_SSHORT`) and 8-bit (`RT_UBYTE` and `RT_SBYTE`) data types.

Example 5-23 uses the `Write_Shared_Varblk_Float(long offset)` syntax.

Example 5-23 Shared Variable Block Generated Code With Callouts (Using the -vbco and -smco Options)

```
/* ------------------------------ Read from Variable */
Enter_Shared_Varblk_Section(4);
   proc2_4_1 = Read_Shared_Varblk_Float(&isi_varblk[0]->block5[0]);
```

```
   proc2_4_2 = Read_Shared_Varblk_Float(&isi_varblk[0]->block5[1]);
Leave_Shared_Varblk_Section(4);

/* ------------------------------ Write to Variable */
Enter_Shared_Varblk_Section(4);
   Write_Shared_Varblk_Float(&isi_varblk[0]->block5[0], proc2_4_1);
   Write_Shared_Varblk_Float(&isi_varblk[0]->block5[1], proc2_4_2);
Leave_Shared_Varblk_Section(4);
```

Global Variable Block Callouts

Callouts are generated around each variable block access when the `-vbco` option is specified. These callouts define a critical section region for the variable block and let you implement some type of exclusion scheme to protect the integrity of the variable block variable access, if necessary. When callouts are used for single- or multiple-processor code generation, the provided implementation of the callouts does absolutely nothing.

Callout Pairs

There are different variations of the callouts to suit different code generation specifications. The callouts are grouped together into pairs. A pair of callouts is used to surround the variable block access. One function of the pair represents the entry into the critical region and the other represents the leaving of the region. There are four different pairs for different code generation configurations. The choice of which pair depends on whether the variable block is shared among multiple processors and whether the `-epi` option is used.

Non-Shared (Local) Global Variable Blocks

A variable block is defined as non-shared if only one processor accesses that variable block. By definition, all variable block accesses for single-processor code generation are non-shared. There are sets of callouts for non-shared variable block accesses. Notice that the term local variable block was previously used to describe a non-shared variable block.

Entering Non-Shared (Local) Critical Section

The prototype of the callout for entering a non-shared global variable block critical section is:

```
void Enter_Local_Varblk_Section(RT_INTEGER index);
procedure Enter_Local_Varblk_Section(index :
RT_INTEGER);
```

The formal argument represents the global reference number for which the variable block is being accessed. The default implementation of those simply calls the `Disable( )` function.

Leaving Non-Shared (Local) Critical Section

The prototype of the callout for leaving a non-shared global variable block critical section is:

```
void Leave_Local_Varblk_Section(RT_INTEGER index);
procedure Leave_Local_Varblk_Section(index :
RT_INTEGER);
```

The formal argument represents the global reference number for which the variable block is being accessed. The default implementation of those simply calls the `Enable( )` function.

The following code uses the `Enter_Local_Varblk...` syntax to call non-shared global variable block generated code with callouts, using the `-vbco` option.

```
Enter_Local_Varblk_Section(4);
       proc2_4_1 = block5[0];
       proc2_4_2 = block5[1];
Leave_Local_Varblk_Section(4);
```

Entering with Extended Procedure Info Option Specified

The prototype of the callout for entering a non-shared global variable block critical section with the extended procedure info option is:

```
void Enter_Local_Varblk_Section(RT_INTEGER index, RT_INTEGER caller_id);
procedure Enter_Local_Varblk_Section(index : RT_INTEGER;
                                          caller_id : RT_INTEGER);
```

The formal argument, `index`, represents the global reference number for which the variable block is being accessed. The second formal argument, `caller_id`, represents a unique identifier for the caller. The default implementation of those calls the `Disable` function.

Leaving with Extended Procedure Info Option Specified

The prototype of the callout for leaving a non-shared global variable block critical section with the extended procedure info option is:

```
void Leave_Local_Varblk_Section(RT_INTEGER index, RT_INTEGER caller_id);
procedure Leave_Local_Varblk_Section(index : RT_INTEGER;
                                          caller_id : RT_INTEGER);
```

The formal argument represents the global reference number for which the variable block is being accessed. The second formal argument, `caller_id`, represents a unique identifier for the caller. The default implementation of those simply calls the `Enable( )` function.

The following code uses the `Enter_Local_Varblk...` syntax to call non-shared global variable block generated code with callouts, using the `-vbco` and `-epi` options.

```
Enter_Local_Varblk_Section(4, 1);
       proc2_4_1 = block5[0];
       proc2_4_2 = block5[1];
Leave_Local_Varblk_Section(4, 1);
```

Shared Global Variable Blocks

Variable blocks are defined as shared if more than one processor accesses that variable block. There are sets of callouts for shared variable block accesses. For a discussion of code generation for shared variable blocks, refer to the *Shared Variable Block Support* section.

Entering Shared Critical Section

The prototype of the callout for entering a shared variable block critical section is:

```
void Enter_Shared_Varblk_Section(RT_INTEGER processor, RT_INTEGER index);
procedure Enter_Shared_Varblk_Section(processor : RT_INTEGER;
                                      index : RT_INTEGER);
```

The first formal argument represents which processor the access is taking place on. Processor numbers are 1-based. The second formal argument represents the global reference number for which the variable block is being accessed.

Leaving Shared Critical Section

The prototype of the callout for leaving a local variable block critical section is:

```
void Leave_Shared_Varblk_Section(RT_INTEGER processor, RT_INTEGER index);
procedure Leave_Shared_Varblk_Section(processor : RT_INTEGER;
                                      index : RT_INTEGER);
```

The first formal argument represents which processor the access is taking place on. Processor numbers are 1-based. The second formal argument represents the global reference number for which the variable block is being accessed.

The following code uses the `Enter_Shared_Varblk...` syntax to call shared variable block generated code with callouts, using the `-vbco` option.

```
Enter_Shared_Varblk_Section(1, 4);
       proc2_4_1 = block5[0];
       proc2_4_2 = block5[1];
Leave_Shared_Varblk_Section(1, 4);
```

Entering with Extended Procedure Info Option Specified

The prototype of the callout for entering a shared variable block critical section with the extended procedure info option is:

```
void Enter_Shared_Varblk_Section(RT_INTEGER index, RT_INTEGER caller_id);
procedure Enter_Shared_Varblk_Section(index : RT_INTEGER;
                                      caller_id : RT_INTEGER);
```

The formal argument, `index`, represents the global reference number for which the variable block is being accessed. The second formal argument, `caller_id`, represents a unique identifier for the caller.

Note A default implementation is not provided for these callouts.

Leaving with Extended Procedure Info Option Specified

The prototype of the callout for leaving a shared variable block critical section with the extended procedure info option is:

```
void Leave_Shared_Varblk_Section(RT_INTEGER index, RT_INTEGER caller_id);
procedure Leave_Shared_Varblk_Section(index : RT_INTEGER;
                                      caller_id : RT_INTEGER);
```

The first formal argument represents the global reference number for which the variable block is being accessed. The second formal argument, `caller_id`, represents a unique identifier for the caller.

Note A default implementation is not provided for these callouts.

The following code uses the `Enter_Shared_Varblk...` syntax for shared variable block generated code with callouts, using the `-vbco` and `-epi` options.

```
Enter_Shared_Varblk_Section(4, 1);
       proc2_4_1 = isi_varblk[0]->block5[0];
       proc2_4_2 = isi_varblk[0]->block5[1];
Leave_Shared_Varblk_Section(4, 1);
```

Caution It is not possible to mix code with shared variable blocks generated with the `-epi` option and code with shared variable blocks generation without `-epi` because the prototypes of the shared variable block callouts are the same.

6

Vectorized Code Generation

This chapter discusses various ways to generate vectorized code. This includes describing the options available, design guidelines, and implementation details about the vectorized code.

Introduction

AutoCode has the capability to generate vectorized code. The default code generation style, however, remains to be all scalars. Vectorized code has two attributes:

- Signals are represented as arrays instead of scalars in the generated code
- Algorithms use loops to roll the code into a compact algorithm

Some of the benefits of vectorized code generation include:

- Smaller source size
- Smaller object code size
- Efficient implementation of large systems
- Loops with general BlockScript blocks
- Mixed scalar and vectorized code within the same model

You do not have to change your pre-release 7.*x* SystemBuild models to get the benefits of vectorization. AutoCode implements all of the standard block algorithms in a vectorized way that can be realized in MATRIXx 7.*x* and later. Of course, a model designed specifically to take advantage of vectorized code will perform better than one designed for scalar code.

How Code Is Generated

This section introduces the look of vectorized code by comparing the code to the equivalent scalar code. The gain block illustrates most of the concepts of vectorized code generation. For this example, assume a 10 input/output gain block with various gain parameters connected directly to the subsystem external input and output.

Scalar Gain Block Example

Example 6-1 shows the scalar code generated for a gain block.

Example 6-1 Scalar Code Generated for Gain Block Example

```
void subsys_1(U, Y)
   struct _Sys_ExtIn *U;
   struct _Subsys_1_out *Y;
{
     static RT_INTEGER iinfo[4];

      /******* Initialization. *******/
      if (SUBSYS_PREINIT[1]) {
         iinfo[0] = 0;
         iinfo[1] = 1;
         iinfo[2] = 1;
         iinfo[3] = 1;
         SUBSYS_PREINIT[1] = FALSE;
         return;
      }

     /***** Output Update. *****/
      /* ----------------------------  Gain Block */
      /* {gain..2} */
      Y->gain_2_1 = 1.2*U->gain_1;
      Y->gain_2_2 = 2.3*U->gain_2;
      Y->gain_2_3 = 3.4*U->gain_3;
      Y->gain_2_4 = 4.5*U->gain_4;
      Y->gain_2_5 = 5.6*U->gain_5;
      Y->gain_2_6 = 6.7*U->gain_6;
      Y->gain_2_7 = 7.8*U->gain_7;
      Y->gain_2_8 = 8.9*U->gain_8;
      Y->gain_2_9 = 9.1*U->gain_9;
      Y->gain_2_10 = 10.11*U->gain_10;
```

The scalar code for the gain block should be familiar. Some characteristics of the code for later comparison should be mentioned. First, this is the canonical example of the concept of unrolling. The basic equation for a gain block is:

$$Y(i) = GainParameter(i) * U(i)$$

where

Y(*i*) is the *i*th output

U(*i*) is the *i*th input

GainParameter(*i*) is the *i*th gain value

i is the range <1..*x*>, where *x* is the number of outputs of the block

As you can see from the code generation, each of the *i*th elements of the gain block is represented by a uniquely named variable. Therefore, the basic equation is considered to be unrolled or code generated with scalars.

The second characteristic is the gain parameters. As you can see, those values are hard-coded into the generated code instead of being represented symbolically. With scalar code generation, constants are always hard-coded.

Vectorized Gain Block Example

Before showing the code for the vectorized gain block, you need to know what to look for and compare against the scalar code. First, look at the implementation of the gain block in Example 6-2. Notice that the code is rolled into a single for-loop. Also notice the use of arrays to access the data and that gain parameter values are no longer hard-coded. The values appear in the ubiquitous `R_P` array.

Example 6-2 Vectorized Code Generation for Gain Block Example

```
void subsys_1(U, Y)
   struct _Sys_ExtIn *U;
   struct _Subsys_1_out *Y;
{
      static RT_INTEGER iinfo[4];

      /***** Parameters. *****/
      static RT_FLOAT R_P[10];
      RT_INTEGER cnt;
      static const RT_FLOAT _R_P[10] = {1.2, 2.3, 3.4, 4.5, 5.6, 6.7,
                                        7.8, 8.9, 9.1, 10.11};

      /***** Algorithmic Local Variables. *****/
      RT_INTEGER i;

      /******* Initialization. *******/
```

```
if (SUBSYS_PREINIT[1]) {
    iinfo[0] = 0;
    iinfo[1] = 1;
    iinfo[2] = 1;
    iinfo[3] = 1;
    for( cnt=0;cnt<10;cnt++ ) {
       R_P[cnt] = _R_P[cnt];
    }
    SUBSYS_PREINIT[1] = FALSE;
    return;
 }

 /***** Output Update. *****/
 /* ---------------------------  Gain Block */
 /* {gain..2} */
 for (i=1; i<=10; i++) {
    Y->gain_2_1[-1+i] = R_P[-1+i]*U->gain_1[-1+i];
 }
```

In summary, Example 6-2 shows some of the requirements for vectorized code generation:

- The block's input and output signals must be arrays.
- Constant parameter data must be represented in an array.
- An array must represent signals of the same data type.

As far as the constant parameter data, AutoCode automatically bundles the data into the `R_P` (or other parameter array) as needed. However, the inputs and outputs of the block are controlled by the signal connectivity of the model. AutoCode will never reconnect your model to provide vectorization.

Array Subscripts

An interesting characteristic of the vectorized code generation is the way array subscripts are generated. Looking closely, you will see in Example 6-2 that all subscripts are generated as `-1+i`. C arrays are 0-based, meaning that the first array element is the 0th element. BlockScript, our internal language to describe block algorithms, defines arrays as being 1-based.[1] Therefore, when a block is translated into code, a translation from 1-based access into 0-based access is performed. In

[1] BlockScript was previously used to generate FORTRAN code in which arrays are 1-based.

addition to issues with the standard block library, all general BlockScript Blocks within the diagram are implemented as 1-based arrays. If the subscript can be evaluated at generation-time, the 0-based subscript will be used.

Note The extra computation of the subscripts represents a compatibility issue. Elimination of the 1-based array representation in BlockScript will be addressed in a future release.

Signal Connectivity

One of the greatest modeling capabilities provided in the SystemBuild model is the ability to connect any output signal to many input pins. That requirement, coupled with increased traceability, necessitated the use of scalars in the generated code. Vectorized code generation requires a more disciplined design if vectorization is to improve the generated code. Although it is not necessary to redesign your model, you should be aware that the flexibility of the SystemBuild Editor lets you design models that vectorize poorly, which means they require the code to be unrolled. In other words, generating arrays is not enough for vectorized code; the connectivity of the model is vital in enabling loops in the generated code.

Block Outputs

The outputs of a block always can be generated as one or more arrays. The outputs of one block cannot be part of the same array used for outputs of another block. The creation of the arrays are controlled by the vectorization mode and labels of the individual output pins.

Block Inputs

The inputs of a block are either SuperBlock (subsystem) external inputs and/or outputs from other basic blocks. Those inputs might or might not be an array and/or might not be connected in a way that allows for the code to be rolled into a loop. The diagram in Figure 6-1 represents poor input connectivity that prevents a rolled loop. Refer to Example 6-3.

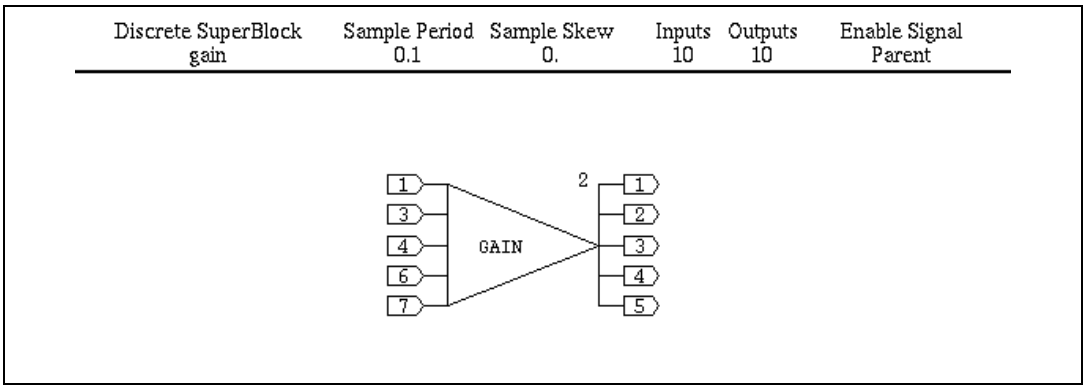


Figure 6-1. Poorly Connected Gain Block

Example 6-3 Generated Code for Poorly Connected Gain Block (for Figure 6-1)

```
void subsys_1(U, Y)
   struct _Subsys_1_in *U;
   struct _Subsys_1_out *Y;
{
      static RT_INTEGER iinfo[4];

      /***** Parameters. *****/
      static RT_FLOAT R_P[5];
      RT_INTEGER cnt;
      static const RT_FLOAT _R_P[5] = {1.2, 2.3, 3.4, 4.5, 5.6};

      /***** Algorithmic Local Variables. *****/
      RT_INTEGER i;

      /******* Initialization. *******/
      if (SUBSYS_PREINIT[1]) {
         iinfo[0] = 0;
         iinfo[1] = 1;
         iinfo[2] = 1;
         iinfo[3] = 1;
         for( cnt=0;cnt<5;cnt++ ) {
            R_P[cnt] = _R_P[cnt];
         }
         SUBSYS_PREINIT[1] = FALSE;
         return;
      }

      /***** Output Update. *****/
```

```
/* ---------------------------  Gain Block */
/* {gain..2} */
Y->gain_2_1[0] = 1.2*U->gain_1[0];
Y->gain_2_1[1] = 2.3*U->gain_1[2];
Y->gain_2_1[2] = 3.4*U->gain_1[3];
Y->gain_2_1[3] = 4.5*U->gain_1[5];
Y->gain_2_1[4] = 5.6*U->gain_1[6];
```

This example shows that the penalty for poor connectivity can be great. In this case, the vectorized code is not an improvement over scalar code.

Vectorization Modes

AutoCode supports two vectorization modes in addition to the default scalar code generation. All three modes are controlled by one command-line option, `-Ov` *n*, where *n* is the mode. The vectorization modes allow the same model to be code generated in several ways to suite your particular goals. The following sections briefly describe each mode.

Maximal Vectorization

Maximal vectorization (mode `-Ov 2`) is one of two vectorization modes supported by AutoCode. Maximal vectorization is defined by placing all of the outputs of a block into one or more arrays. For most blocks, only one array is needed because the block can only have one output data type. For blocks with more than one output data type, more than one array is used. External inputs also are formed into arrays and like the basic block, if mixed data types are used, multiple arrays are generated.

The names of the arrays are taken from the label/name of the first signal bundled into the array. As a result, maximal vectorization might not produce generated code that is very traceable back to the diagram. This vectorization mode is to provide a quick way to get vectorized code without having to examine your model's design and tune it for efficient code.

Mixed Vectorization

This vectorization mode (mode `-Ov 1`) allows for both scalar and vector code generation within the same system. This mode also is called vector-by-label because the labels/names of the signals determine if a vector is generated.

Vector Labels and Names

The SuperBlock Editor lets you give a name to a range of signals using a special notation. The Editor then repeatedly applies that name to the range. AutoCode interprets this type of labeling as a definition of which pins are to be an array. For more information about vector labeling, refer to the *SystemBuild User Guide*.

The editor lets you use a vector label more liberally than what makes sense for code generation. The following restrictions apply to a vector label when being translated into code. If the labeling of the diagram does not conform, AutoCode creates the arrays as best it can by mangling the name where it sees fit.

- A vector must start at index 1.
- A vector cannot span different data types.
- A vector must be a range of contiguous pins.
- A vector can only be defined from one block, either a basic block's outputs, top-level SuperBlock inputs, or subsystem/procedure boundary.
- A signal with an empty label/name is generated as a scalar.

Example

Figure 6-2 shows a good example of a vectorized algorithm, although it does not do anything significant. Both the maximal and mixed-mode code generation also is provided in Example 6-4 and Example 6-5, respectively. Notice that the diagram is shown with all labels shown for each pin—even those pins with a vector label—so you can trace the signals within the code.

In the model, notice that the outputs of the gain block are using scalar labels. Therefore, when generated for maximal Vectorization, the array will be named Throttle. When generated in mixed mode, you will see all five distinct outputs of that block. Notice how that choice prevents the gain block and time delay block from vectorizing. Notice that in either mode, block states are always a vector.

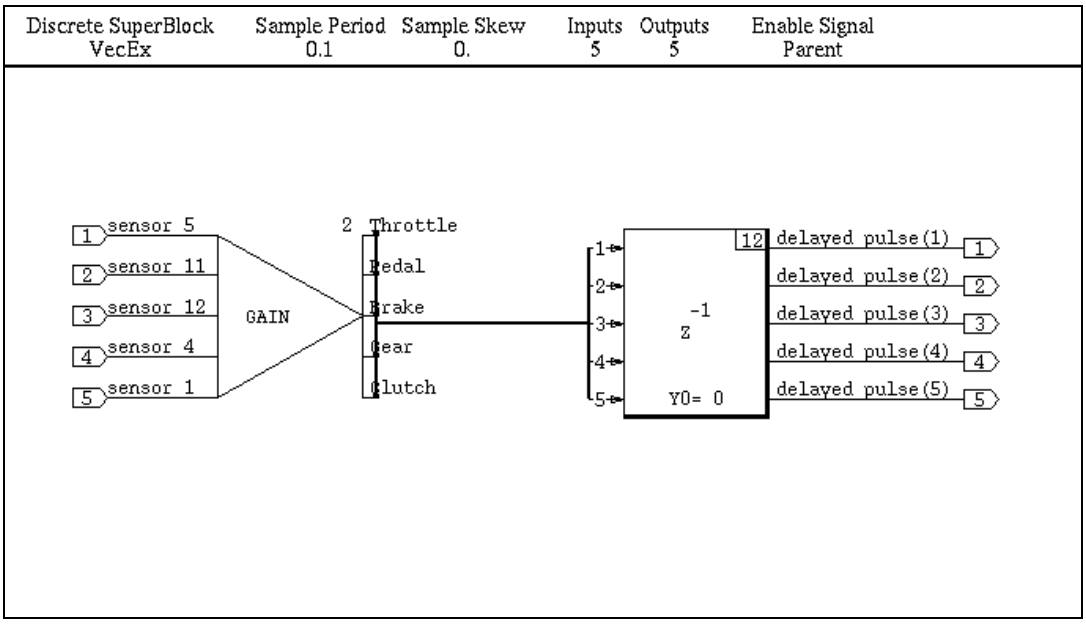


Figure 6-2. Example Model Diagram

Example 6-4 Maximal Vectorized Code Generation (for Figure 6-2)

```
/******* System Ext I/O type declarations. *******/
struct _Subsys_1_out {
   RT_FLOAT delayed_pulse[5];
};

struct _Sys_ExtIn {
   RT_FLOAT sensor_5[5];
};

/***** States type declaration. *****/
struct _Subsys_1_states {
   RT_FLOAT sensor_delay[5];
};

void subsys_1(U, Y)
   struct _Sys_ExtIn *U;
   struct _Subsys_1_out *Y;
{
      /***** States Array. *****/
      static struct _Subsys_1_states ss_1_states[2];
```

```
static struct _Subsys_1_states *X;
static struct _Subsys_1_states *XD;
static struct _Subsys_1_states *XTMP;
static RT_INTEGER iinfo[4];
static RT_INTEGER INIT;

/***** Parameters. *****/
static RT_FLOAT R_P[11];
RT_INTEGER cnt;
static const RT_FLOAT _R_P[11] = {0.1, 0.0, 0.0, 0.0, 0.0,
                                  0.0, -8.7, -7.6, -6.5,
                                  -5.4, -4.3};

/***** Local Block Outputs. *****/
RT_FLOAT Throttle[5];

/***** Algorithmic Local Variables. *****/
RT_INTEGER i;
RT_INTEGER j;
RT_INTEGER k;

/******* Initialization. *******/
if (SUBSYS_PREINIT[1]) {
   iinfo[0] = 0;
   iinfo[1] = 1;
   iinfo[2] = 1;
   iinfo[3] = 1;
   INIT = 1;
   X = &ss_1_states[0];
   XD = &ss_1_states[1];
   {
      RT_INTEGER ii;
      for( ii=0;ii<5;ii++ ) {
         X->sensor_delay[ii] = 0.0;
      }
   }
   {
      RT_INTEGER ii;
      for( ii=0;ii<5;ii++ ) {
         XD->sensor_delay[ii] = 0.0;
      }
   }
   for( cnt=0;cnt<11;cnt++ ) {
      R_P[cnt] = _R_P[cnt];
```

```
        }
        SUBSYS_PREINIT[1] = FALSE;
        return;
    }

    /***** Output Update. *****/
    /* ----------------------------  Time Delay */
    /* {VecEx..12} */
    if (INIT) {
        k = 0;
        for (i=1; i<=5; i++) {
            X->sensor_delay[k] = R_P[i];
            k = k + 1;
        }
    }
    k = 1;
    for (i=1; i<=5; i++) {
        Y->delayed_pulse[-1+i] = X->sensor_delay[-1+k];
        k = k + 1;
    }
    /* ----------------------------  Gain Block */
    /* {VecEx..2} */
    for (i=1; i<=5; i++) {
        Throttle[-1+i] = R_P[5+i]*U->sensor_5[-1+i];
    }

    /***** State Update. *****/
    /* ----------------------------  Time Delay */
    /* {VecEx..12} */
    k = 0;
    for (i=1; i<=5; i++) {
        XD->sensor_delay[k] = Throttle[-1+i];
        k = k + 1;
    }

    /***** Swap state pointers. *****/
    XTMP = X;
    X = XD;
    XD = XTMP;
    INIT = 0;
}
```

Example 6-5 Mixed Vectorized Code Generation (for Figure 6-2)

```
/******* System Ext I/O type declarations. *******/
struct _Subsys_1_out {
   RT_FLOAT delayed_pulse[5];
};

struct _Sys_ExtIn {
   RT_FLOAT sensor_5;
   RT_FLOAT sensor_11;
   RT_FLOAT sensor_12;
   RT_FLOAT sensor_4;
   RT_FLOAT sensor_1;
};

/***** States type declaration. *****/
struct _Subsys_1_states {
   RT_FLOAT sensor_delay[5];
};

void subsys_1(U, Y)
   struct _Sys_ExtIn *U;
   struct _Subsys_1_out *Y;
{
      /***** States Array. *****/
      static struct _Subsys_1_states ss_1_states[2];

      /***** Current and Next States Pointers. *****/
      static struct _Subsys_1_states *X;
      static struct _Subsys_1_states *XD;
      static struct _Subsys_1_states *XTMP;
      static RT_INTEGER iinfo[4];
      static RT_INTEGER INIT;

      /***** Parameters. *****/
      static RT_FLOAT R_P[11];
      RT_INTEGER cnt;
      static const RT_FLOAT _R_P[11] = {0.1, 0.0, 0.0, 0.0, 0.0,
                                        0.0, -8.7, -7.6, -6.5,
                                        -5.4, -4.3};

      /***** Local Block Outputs. *****/
      RT_FLOAT Throttle;
      RT_FLOAT Pedal;
```

```
RT_FLOAT Brake;
RT_FLOAT Gear;
RT_FLOAT Clutch;

/***** Algorithmic Local Variables. *****/
RT_INTEGER i;
RT_INTEGER j;
RT_INTEGER k;

/******* Initialization. *******/
if (SUBSYS_PREINIT[1]) {
   iinfo[0] = 0;
   iinfo[1] = 1;
   iinfo[2] = 1;
   iinfo[3] = 1;
   INIT = 1;
   X = &ss_1_states[0];
   XD = &ss_1_states[1];
   {
      RT_INTEGER ii;
      for( ii=0;ii<5;ii++ ) {
         X->sensor_delay[ii] = 0.0;
      }
   }
   {
      RT_INTEGER ii;
      for( ii=0;ii<5;ii++ ) {
         XD->sensor_delay[ii] = 0.0;
      }
   }
   for( cnt=0;cnt<11;cnt++ ) {
      R_P[cnt] = _R_P[cnt];
   }
   SUBSYS_PREINIT[1] = FALSE;
   return;
}

/***** Output Update. *****/
/* ---------------------------  Time Delay */
/* {VecEx..12} */
if (INIT) {
   k = 0;
   for (i=1; i<=5; i++) {
      X->sensor_delay[k] = R_P[i];
```

```
      k = k + 1;
   }
}
k = 1;
for (i=1; i<=5; i++) {
   Y->delayed_pulse[-1+i] = X->sensor_delay[-1+k];
   k = k + 1;
}
/* ----------------------------  Gain Block */
/* {VecEx..2} */
Throttle = -8.7*U->sensor_5;
Pedal = -7.6*U->sensor_11;
Brake = -6.5*U->sensor_12;
Gear = -5.4*U->sensor_4;
Clutch = -4.3*U->sensor_1;

/***** State Update. *****/
/* ----------------------------  Time Delay */
/* {VecEx..12} */
k = 0;
XD->sensor_delay[k] = Throttle;
k = k + 1;
XD->sensor_delay[k] = Pedal;
k = k + 1;
XD->sensor_delay[k] = Brake;
k = k + 1;
XD->sensor_delay[k] = Gear;
k = k + 1;
XD->sensor_delay[k] = Clutch;
k = k + 1;

/***** Swap state pointers. *****/
XTMP = X;
X = XD;
XD = XTMP;
INIT = 0;
```

Vectorization Features

This section describes features of the vectorized code. You do not have to enable these features explicitly. These features are the natural result of introducing arrays into the generated code. In other words, to support vectorization for any model, these features must be present.

Note The examples within this section assume maximal vectorization unless otherwise noted.

Multiple Arrays within a Block

All blocks support multiple vectors (arrays) as both outputs and inputs. However, depending on exactly how the signals are connected and the algorithm, loops might not be generated as expected. Generally speaking, as long as multiple arrays are connected contiguously to the inputs of the block, loops are possible. For example, examine the model in Figure 6-3 and the generated code in Example 6-6.

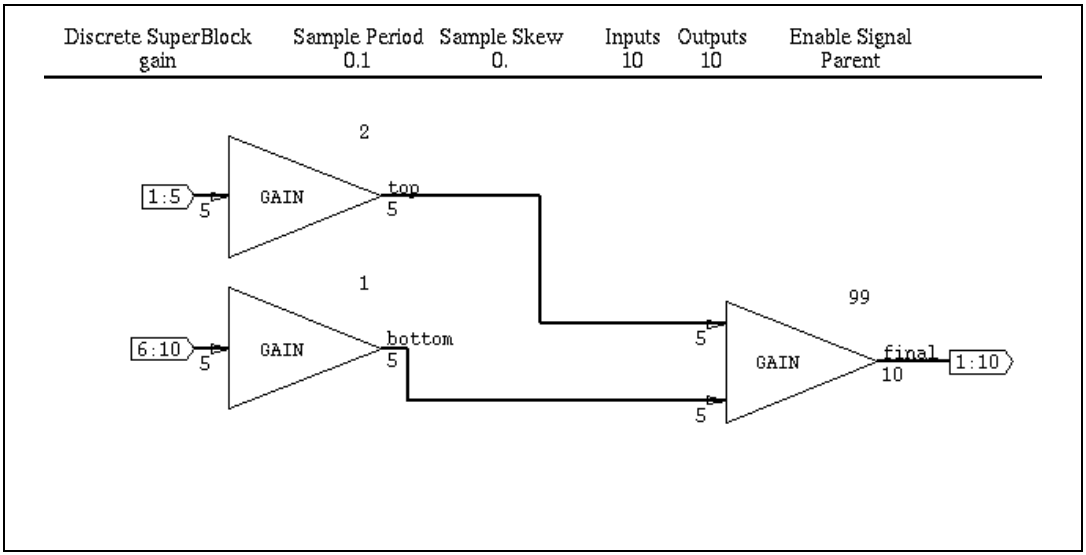


Figure 6-3. Multiple Array Mode

Example 6-6 Multiple Array Code (for Figure 6-3)

```
void subsys_1(U, Y)
   struct _Sys_ExtIn *U;
   struct _Subsys_1_out *Y;
{
      static RT_INTEGER iinfo[4];

      /***** Parameters. *****/
      static RT_FLOAT R_P[20];
      RT_INTEGER cnt;
      static const RT_FLOAT _R_P[20] = {1.2, 2.3, 3.4, 4.5, 5.6,
                                        1.2, 2.3, 3.4, 4.5, 5.6,
```

```
                                        1.2, 2.3, 3.4, 4.5, 5.6,
                                        1.0, 1.0, 1.0, 1.0, 1.0};

/***** Local Block Outputs. *****/
RT_FLOAT top[5];
RT_FLOAT bottom[5];

/***** Algorithmic Local Variables. *****/
RT_INTEGER i;

/******* Initialization. *******/
if (SUBSYS_PREINIT[1]) {
   iinfo[0] = 0;
   iinfo[1] = 1;
   iinfo[2] = 1;
   iinfo[3] = 1;
   for( cnt=0;cnt<20;cnt++ ) {
      R_P[cnt] = _R_P[cnt];
   }
   SUBSYS_PREINIT[1] = FALSE;
   return;
}

/***** Output Update. *****/
/* ----------------------------  Gain Block */
/* {gain..2} */
for (i=1; i<=5; i++) {
   top[-1+i] = R_P[-1+i]*U->gain_1[-1+i];
}
/* ----------------------------  Gain Block */
/* {gain..1} */
for (i=1; i<=5; i++) {
   bottom[-1+i] = R_P[4+i]*U->gain_1[4+i];
}
/* ----------------------------  Gain Block */
/* {gain..99} */
for (i=1; i<=5; i++) {
   Y->final[-1+i] = R_P[9+i]*top[-1+i];
}
for (i=6; i<=10; i++) {
   Y->final[-1+i] = R_P[9+i]*bottom[-6+i];
}}
```

The interesting part is that of the last gain block (`gain..99`). Notice that although two distinct arrays are used as input, because the input arrays are connected contiguously, the code is rolled into two separate loops. The generalized capability can be thought of as a replication of the block algorithm to produce the best vectorization based on the inputs, outputs, and the block algorithm.

Note There can be more than one array for the outputs of a block just as there can be multiple arrays for the inputs. AutoCode replicates the algorithm to support multiple output arrays just as Example 6-6 showed with multiple inputs.

Split-Merge Inefficiency

The term split-merge is a description of a problem that occurs when generating vectorized code for a diagram and pieces of one or more arrays are used as input. This is called a *split* [input] vector problem. A split vector can prevent blocks from rolling into a single loop. A solution to this problem is the merging of all of the inputs into arrays that can be easily rolled.

Split Vector

The SystemBuild Editor lets you connect individual pins of the blocks very easily. Thus, when generating vectorized code, instead of connecting up the whole array, pieces or slices of the array are being used. This does not pose a problem for the block generating the data, but for the blocks consuming the data—that is, blocks using part of the array as input. The problem is that the generated code might not be rolled into a loop. Hence, one would say that you are using a split vector.

Consider the diagram in Figure 6-4 and the generated code in Example 6-7.

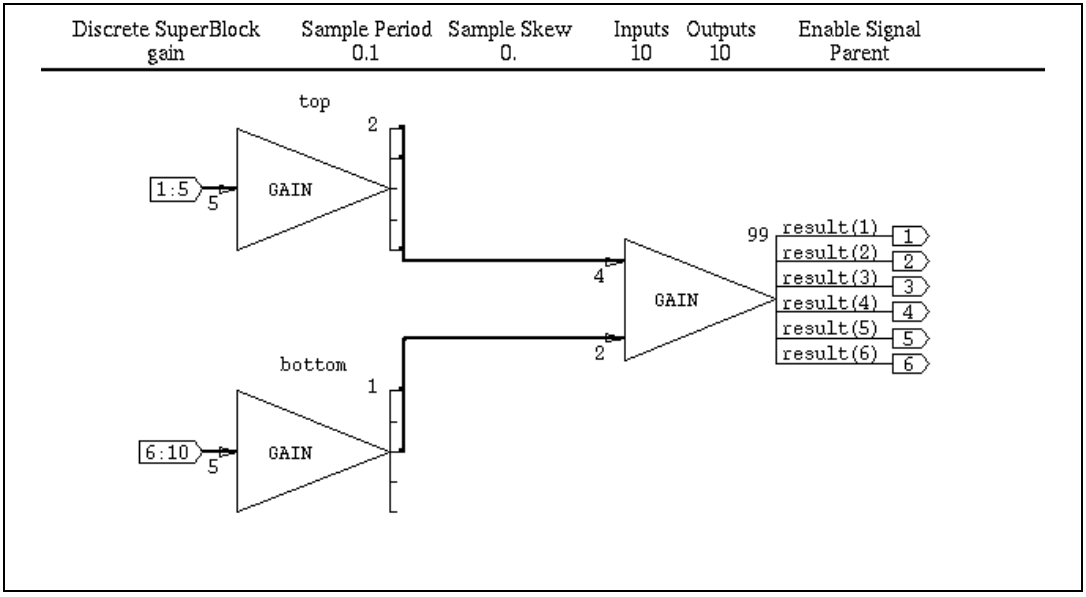


Figure 6-4. Example of a Split Vector

Example 6-7 Generated Code for Split Vector (for Figure 6-4)

```
/***** Output Update. *****/
/* ----------------------------  Gain Block */
/* {gain.top.2} */
for (i=1; i<=5; i++) {
   top[-1+i] = R_P[-1+i]*U->gain_1[-1+i];
}
/* ----------------------------  Gain Block */
/* {gain.bottom.1} */
for (i=1; i<=5; i++) {
   bottom[-1+i] = R_P[4+i]*U->gain_1[4+i];
}
/* ----------------------------  Gain Block */
/* {gain..99} */
for (i=1; i<=2; i++) {
   Y->result[-1+i] = R_P[9+i]*top[-4+4*i];
}
Y->result[2] = 3.4*bottom[2];
Y->result[3] = 4.5*top[1];
Y->result[4] = 5.6*bottom[0];
Y->result[5] = 6.7*top[1];}
```

The two producer gain blocks (top, bottom) vectorize as expected. The (gain..99) does not vectorize well. Notice that AutoCode was only able to vectorize two inputs while the remaining four were unrolled.

AutoCode does not attempt to reconnect your diagram to generate better vectorized code. Two design-level solutions can be applied to eliminate the split vector problem:

- Change the design so that the inputs naturally form the array, that is, merging multiple blocks into one block.
- Change to introduce blocks that merge data.

Merge

Merge is effectively a copy of sparsely connected data into a single array. There is no special block to perform this because the gain block with unity gain parameters performs this task perfectly. Consider the model shown in Figure 6-5 that is similar to Example 6-7, and the generated code in Example 6-8.

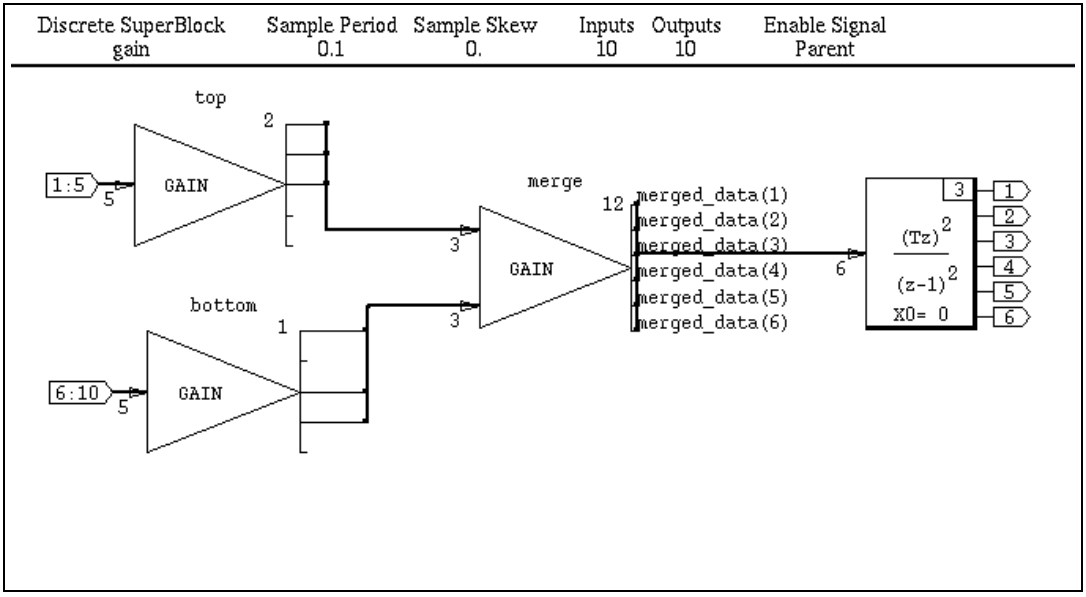


Figure 6-5. Complete Split-Merge Diagram

Example 6-8 Generated Code (for Figure 6-5)

```
/***** Output Update. *****/
/* ----------------------------  Gain Block */
/* {gain.top.2} */
for (i=1; i<=5; i++) {
   top[-1+i] = R_P[-1+i]*U->gain_1[-1+i];
}
/* ----------------------------  Gain Block */
/* {gain.bottom.1} */
for (i=1; i<=5; i++) {
   bottom[-1+i] = R_P[4+i]*U->gain_1[4+i];
}
/* ----------------------------  Gain Block */
/* {gain.merge.12} */
merged_data[0] = top[2];
merged_data[1] = bottom[3];
merged_data[2] = top[0];
merged_data[3] = bottom[2];
merged_data[4] = top[1];
merged_data[5] = bottom[0];
/* ----------------------------  Nth Order Integrator */
/* {gain..3} */
if (INIT && XREMAP) {
   for (i=1; i<=6; i++) {
      X->gain_3_S1[0] = X->gain_3_S1[0] - TSAMP*X->gain_3_S1[1];
      X->gain_3_S1[1] = X->gain_3_S1[1] - TSAMP*merged_data[1+i];
   }
}
for (i=1; i<=6; i++) {
   Y->gain_3_1[-1+i] = X->gain_3_S1[1] + TSAMP*merged_data[-1+i];
   Y->gain_3_1[-1+i] = X->gain_3_S1[0] + TSAMP*Y->gain_3_1[-1+i];
   Y->gain_3_1[-1+i] = R_P[15+i]*Y->gain_3_1[-1+i];
}
/***** State Update. *****/
/* ----------------------------  Nth Order Integrator */
/* {gain..3} */
for (i=1; i<=6; i++) {
   XD->gain_3_S1[1] = X->gain_3_S1[1] + TSAMP*merged_data[-1+i];
   XD->gain_3_S1[0] = X->gain_3_S1[0] + TSAMP*(XD->gain_3_S1[1]);}
```

You should notice two things in the code shown in Example 6-8. First, the gain block added to merge the data is generated as copies from the respective inputs into the single array. Second, the integrator block is tightly rolled. If the merge was not present, the Integrator would have been unrolled, causing a 6-fold increase in the amount of code for that block.

The only reason to introduce a merge block (unit gain block) is when the cost of unrolling the algorithm of your block—in this case the integrator block—is more expensive then the merge block. It can be seen from the code that the cost of a merge block is a copy in a local array. Because the integrator algorithm is complicated, it is necessary to have the merge so that the integrator is rolled.

AutoCode will not automatically introduce the merge (copy) just to improve vectorization. The reason is that traceability from the code to model is reduced anytime extra code other than the block algorithm is generated. Also, AutoCode is not able to evaluate the design decision to make one block rolled at the expense of another. Therefore, for optimal vectorization, you might need to change your model.

External Outputs

Another variation of the split-merge problem appears with external outputs. External outputs are represented by the `Y`-structure. It contains only those signals marked as external outputs. For scalar code generation, AutoCode directly uses the symbol in the `Y`-structure instead of using local storage. However, when the output of a block is a vector and only a subset of the outputs are connected to external outputs, a conflict of requirements appears between storing the block output into an array and optimizing access to external output.

Copy-Back

When a split-merge occurs with external outputs, AutoCode must act to preserve the semantics of the model. AutoCode has been designed to preserve the array, and therefore the block vectorization and *copy-back* those external outputs from the array into the `Y`-structure. In the example shown in Figure 6-6, a simple gain block has only two of its five outputs connected to the external output. AutoCode preserves the array for the gain block, but copies the pieces of the array that are external output into the `Y`-structure.

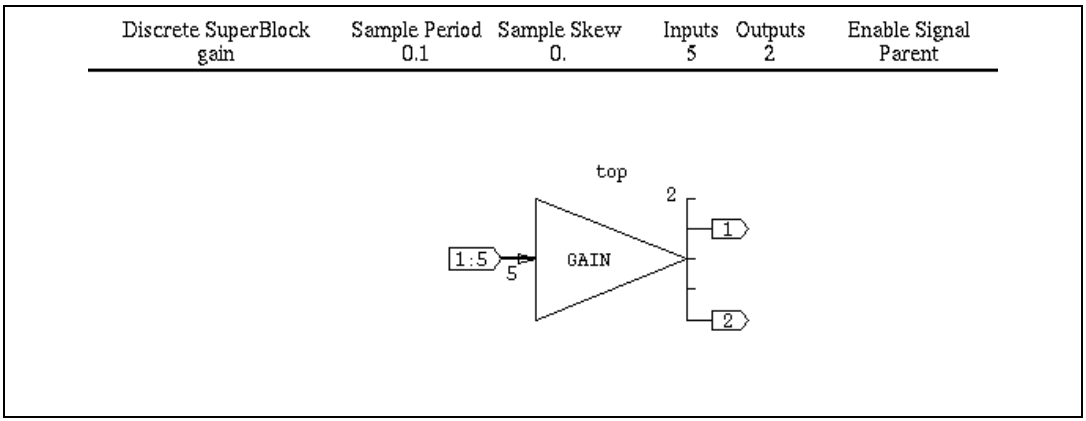


Figure 6-6. Copy-Back Example

Example 6-9 Generated Code (for Figure 6-6)

```
void subsys_1(U, Y)
   struct _Sys_ExtIn *U;
   struct _Subsys_1_out *Y;
{
      /**** some code deleted ****/

      /***** Output Update. *****/
      /* ---------------------------  Gain Block */
      /* {gain.top.2} */
      for (i=1; i<=5; i++) {
         top[-1+i] = R_P[-1+i]*U->gain_1[-1+i];
      }

      /* Copy back(s) and/or duplicate(s) */
      {
         RT_INTEGER k=0;
         for( k=0;k<2;k++ ) {
            Y->top[k] = top[(k*3)+1];
         }
      }
```

Copy-backs are coupled with handling duplicate external outputs, meaning that a block output is connected to more than one external output pin. The example in Figure 6-6 just contains a copy-back. The copy-back/duplicate copies, if needed, will appear at the very end of the subsystem.

Eliminating Copy-Back

There are many design ways to eliminate or hide the extra copies of the copy-back. All of them can be categorized into two groups:

- Use Mixed Vectorization mode and force scalars to be used for the block is sparse external outputs. This eliminates rolling of the block algorithm while eliminating the copy.
- Take all of the outputs of the block as external outputs. This adds more elements to the `Y`-structure, but preserves rolling, and there would be no copy-backs.

Other Copy-Back Scenarios

Copy-backs are most common at the top-level SuperBlock of a single-rate system, and with Standard Procedures external outputs. In a single-rate subsystem or Standard Procedure, it is important to notice that just having all of the block outputs connected to external output will not eliminate the copy-back unless the outputs are connected to *contiguous pins* of the external output. Multi-rate systems are more forgiving, because AutoCode will rearrange the outputs within the `Y`-structure, and thus having all of the block outputs connected to external output is sufficient.

Vectorized Standard Procedure Interface

Another feature of the vectorized generated code is the ability to provide an efficient mechanism to pass data into and out of standard procedures. When generating vectorized code coupled with the `no-uy` procedure interface, AutoCode creates arrays to be passed to the procedures. Passing an array means passing by pointer, and that translates into a significant decrease in the procedure call overhead. Figure 6-7 shows an example of this feature, and Example 6-10 shows the code generated.

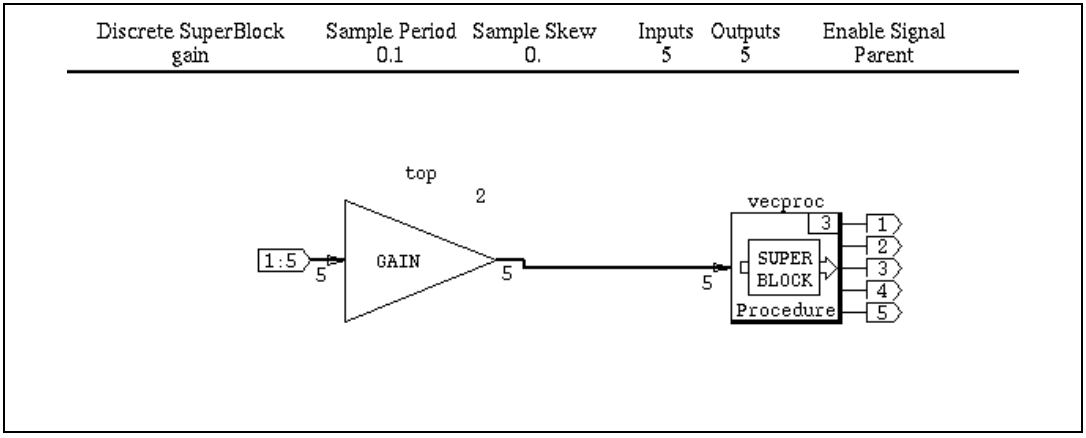


Figure 6-7. Vectorized Procedure Interface

Example 6-10 Generated Code (for Figure 6-7)

```
/******* Procedure: vecproc *******/
void vecproc(vecproc_1, vecproc_2_1, I)
   RT_FLOAT vecproc_1[5];
   RT_FLOAT vecproc_2_1[5];
   struct _vecproc_info *I;
{
      /**** some code deleted ****/

      /***** Output Update. *****/
      /* ---------------------------  Gain Block */
      /* {vecproc..2} */
      for (i=1; i<=5; i++) {
         vecproc_2_1[-1+i] = R_P[-1+i]*vecproc_1[-1+i];
      }
}

void subsys_1(U, Y)
   struct _Sys_ExtIn *U;
   struct _Subsys_1_out *Y;
{
      /**** some code deleted ****/

      /***** Output Update. *****/
      /* ---------------------------  Gain Block */
      /* {gain.top.2} */
      for (i=1; i<=5; i++) {
```

```
   top[-1+i] = R_P[-1+i]*U->gain_1[-1+i];
}
/* ----------------------------  Procedure Super Block */
/* {vecproc.3} */
vecproc(&top[0], &Y->vecproc_3_1[0], &vecproc_3_i);
iinfo[0] = vecproc_3_i.iinfo[0];
if( iinfo[0] != 0 ) {
   vecproc_3_i.iinfo[0] = 0; goto EXEC_ERROR;
}
```

A vectorized procedure interface suffers from a form of the copy-back problem. When using the vectorized interface, each time the procedure is called, the inputs and outputs must be an array. Therefore, if an array is not connected as the input or output, AutoCode must generate code to copy data into a temporary array and then pass that array. In other words, if the inputs and/or outputs are not connected to naturally conform to a vector interface, the copies required might add significant overhead to the procedure call, eliminating the benefits of the vectorized interface.

Ada Array Aggregates and Slices

All of the examples presented include generated C code. AutoCode will generate the equivalent Ada code, except that use of array aggregation and slices are used when possible. Example 6-11 is the equivalent Ada code for a diagram previously shown in Figure 6-2.

Note Only the relevant subsystem code is listed.

Example 6-11 Ada Generated Code (for Figure 6-2)

```
procedure subsys_1 is

   ------ Local Block Outputs. ------
   Throttle : RT_FLOAT_AY(0..4);

   ------ Algorithmic Local Variables. ------
   i_1 : RT_INTEGER;
   j : RT_INTEGER;
   k_1 : RT_INTEGER;

begin

   -------- Initialization. --------
   if SUBSYS_PREINIT(1) then
      iinfo(0..3) := (0,1,1,1);
```

```
      INIT := TRUE;
      X := ptr_of(ss_1_states(0)'address);
      XD := ptr_of(ss_1_states(1)'address);
      X.sensor_delay := (others => 0.0);
      XD.sensor_delay := (others => 0.0);
      for cnt in RT_INTEGER range 0..10 loop
         R_P(cnt) := RP(cnt);
      end loop;
      SUBSYS_PREINIT(1) := FALSE;
      return;
   end if;

   ------ Output Update. ------
   -- ----------------------------  Time Delay --
   -- {VecEx..12} --
   if INIT then
      k_1 := 0;
      for i_1 in RT_INTEGER range 1..5 loop
         X.sensor_delay(k_1) := R_P(i_1);
         k_1 := k_1 + 1;
      end loop;
   end if;
   k_1 := 1;
   for i_1 in RT_INTEGER range 1..5 loop
      Y.delayed_pulse(-1+i_1) := X.sensor_delay(-1+k_1);
      k_1 := k_1 + 1;
   end loop;
   -- ----------------------------  Gain Block --
   -- {VecEx..2} --
   for i_1 in RT_INTEGER range 1..5 loop
      Throttle(-1+i_1) := R_P(5+i_1)*U.sensor_5(-1+i_1);
   end loop;

   ------ State Update. ------
   -- ----------------------------  Time Delay --
   -- {VecEx..12} --
   k_1 := 0;
   for i_1 in RT_INTEGER range 1..5 loop
      XD.sensor_delay(k_1) := Throttle(-1+i_1);
      k_1 := k_1 + 1;
   end loop;
```

```
------ Swap state pointers. ------
      XTMP := X;
      X := XD;
      XD := XTMP;
      INIT := FALSE;
end;
```

Vectorization of the BlockScript Block

The general BlockScript block is commonly used to implement very complicated or custom algorithms within a single block. A major limitation was the so-called soft-index limitation that occurred when generating code. The soft-index limitation existed because BlockScript represented inputs and outputs as vectors while AutoCode generated scalars. Therefore, you could not use a run-time computed subscript (that is, soft) index for inputs and outputs.

With vectorization, the soft-subscript limitation is gone, as the inputs and outputs of the block are arrays. That means that the connectivity of the inputs matches the input groups specified in the BlockScript code.

Note Even with arrays, it is still possible to have the soft-subscript limitation for the inputs. The last determining factor is how the inputs to the block are connected. If the inputs to the BlockScript block are scalars rather than an array, the soft-subscript limitation will apply.

The following is a list of limitations and features of the general BlockScript block as applied to vectorized code generation.

- It is an error to use both soft-subscript and scalar code generation.
- Only 8-nested loops are supported.
- Bounds checks are not performed for a soft-subscript array access.
- While loops do not support a soft-subscript for inputs or outputs.
- A soft-subscript expression of an input or output array is not supported outside of a For loop.

For more information about the BlockScript block, refer to the *BlockScript Block* section of Chapter 5, *Generated Code Architecture*.

Matrix Outputs

When you provide matricized output labeling for a block, AutoCode generates the resulting “matrix” as a single-dimensional array, even in Ada. This means that the output of a matrix-labeled block and the output of a vector-labeled block are identical, provided the number of elements in the matrix matches the number in the vector. This paradigm was selected because it insulates the “user” of a signal from the nature of the signal source. Imagine trying to generate code for a block and having to first check whether one or more input arrays are single versus double-dimensional arrays.

One drawback is that some minor arithmetic must often be done in the generated code to calculate the proper index expression into the “matrix.” That is, row and column positions must be flattened to a single-dimensional index number. The performance impact should be minimal and this approach has the benefit that all optimizations originally implemented for vectors now transfer immediately to “matrices,” such as partial reuse of vectorized signals, with the `-Oreuse` option.

There is no rule that limits the use of matrix labeling to the newly introduced matrix blocks. You can use matrix labeling on any block outputs. However, since the matrix labels do not produce two-dimensional arrays in the generated code, and block functionality is not affected by the type of label present, it becomes clear that the main benefit of matrix labeling is in the SystemBuild Editor and Connection Editor. For code generation purposes, you will see no real performance gain in matrix labeling over vector labeling. For information on matrix optimization, refer to the *Optimizing with Matrix Blocks* section of Chapter 7, *Code Optimization*.

7

Code Optimization

This chapter explains the details of generating production quality code for micro controller-based applications. Generally, micro controller-based applications have stringent requirements for code size and execution speed. AutoCode supports various optimizations that could be used effectively to generate highly optimal code both in terms of speed and code size. Chapter 6, *Vectorized Code Generation*, explains the first step in this direction, namely generating vectorized code that could significantly reduce the code size of an application. This chapter explores other optimizations supported by AutoCode.

Read from Variable Blocks

Variable blocks, both global and local, could be extensively used in any application in order to store values from computations. AutoCode sequences variable blocks and uses them in a deterministic fashion. When a Read from Variable block is used, it is first copied into a temporary local variable, and this variable is used in all further computations. This is not strictly required for local variable blocks; and for global variable blocks that are not shared across subsystems. Hence, instead of using the temporary variable, the variable block could be used directly in all the computations. Elimination of the temporary variable would result in reduced code size, reduced stack size, and improved execution speed.

The code fragment in Example 7-1 shows the code generated by AutoCode without this optimization; Example 7-2 shows code generated with variable block optimization.

Example 7-1 Code Generated without Variable Block Optimization

```
/* Model variable definitions */
RT_FLOAT var;
RT_FLOAT var1;

void subsys_1(U,Y)
 struct _Sys_ExtIn *U;
 struct _Subsys_1_out *Y;
{
```

```
 static RT_INTEGER iinfo[4];

 /***** Local Block Outputs. *****/
      RT_FLOAT new_11_1;
      RT_FLOAT new_1_1;
      RT_FLOAT new_12_1;

/******* Initialization. *******/

     if (SUBSYS_PREINIT[1]) {
       iinfo[0] = 0; iinfo[1] = 1;
       iinfo[2] = 1;
       iinfo[3] = 1;
       SUBSYS_PREINIT[1] = FALSE;
       return;
     }

/***** Output Update. *****/
/* ----------------------------  Read from Variable */
/* {new..11} */
new_11_1 = var;
/* ----------------------------  Read from Variable */
/* {new..1} */
new_1_1 = var1;
/* ----------------------------  Sum of Vectors */
/* {new..12} */
new_12_1 = U->new_1 + new_1_1;
/* ----------------------------  Write to Variable */
/* {new..3} */
var = new_12_1;
/* ----------------------------  Sum of Vectors */
/* {new..5} */
Y->new_5_1 = U->new_1 - new_11_1;
if(iinfo[1]) {
   SUBSYS_INIT[1] = FALSE;
  iinfo[1] = 0;
}
return;
EXEC_ERROR: ERROR_FLAG[1] = iinfo[0];
iinfo[0]=0;}
```

Example 7-2 Code Generated with Variable Block Optimization Turned On

```
/* Model variable definitions */
RT_FLOAT var;
RT_FLOAT var1;

void subsys_1(U, Y)
struct _Sys_ExtIn *U;
struct _Subsys_1_out *Y;
{
   static RT_INTEGER iinfo[4];

  /***** Local Block Outputs. *****/
     RT_FLOAT new_11_1;
     RT_FLOAT new_12_1;

  /******* Initialization. *******/

     if (SUBSYS_PREINIT[1]) {
       iinfo[0] = 0;
       iinfo[1] = 1;
       iinfo[2] = 1;
       iinfo[3] = 1;
       SUBSYS_PREINIT[1] = FALSE;
       return;
     }
 /***** Output Update. *****/
 /* ----------------------------  Read from Variable */
 /* {new..11} */
 new_11_1 = var;
 /* ----------------------------  Read from Variable */
 /* {new..1} */

 /* ----------------------------  Sum of Vectors */
 /* {new..12} */
 new_12_1 = U->new_1 + var1;
 /* ----------------------------  Write to Variable */
 /* {new..3} */
 var = new_12_1;
 /* ----------------------------  Sum of Vectors */
 /* {new..5} */
 Y->new_5_1 = U->new_1 - new_11_1;
  if (iinfo[1]) {
    SUBSYS_INIT[1] = FALSE;
```

```
  iinfo[1] = 0;
}
return;
EXEC_ERROR: ERROR_FLAG[1] = iinfo[0];
iinfo[0]=0;}}
```

AutoCode performs this optimization only if it is safe to do so. There could be circumstances that could potentially prevent this optimization from taking place. AutoCode looks for the presence of a write-to-the-same-variable block between the Read from Variable block statement and its use in later computations. AutoCode does not optimize variable blocks under these circumstances, as doing so would change the meaning of the program.

The subsystem code in Example 7-1 actually uses two global variable blocks, namely `var` and `var1`. Example 7-2 shows the code generated with global variable block optimization turned on. While variable block `var1` is used directly in the summation block computation, an extra variable `new_11_1` is used for the variable block `var`. This is because there is a Write to Variable block (modifying `var`) present in between the Read from Variable block and its use in a summation block.

Optimization of local and global variables can be controlled independently. The command-line arguments `-Olvarblk` and `-Ogvarblk` bring about the optimization of extra variables associated with the Read from local and global variable blocks, respectively.

Notice that the global variable block optimization will work only if they are not shared between subsystems or only if the `-vbco` option is not used.

The following items could prevent AutoCode from optimizing Read from Variable blocks.

- Presence of Write to the same variable block in between a Read from Variable block and its use.
- Existence of a Procedure SuperBlock (Writing to the variable block) between the Read from Variable block and its use.
- A dynamic block is connected to a Read from Variable block, and there is a Write to same variable block in the subsystem (or procedure). Since state UPDATE is done toward the end of the subsystem, the Write to Variable block comes in between the Read from Variable block and its use (in the state update).
- A Read from Variable block is outside a while loop, and the block reading this variable block is inside the loop. The loop also contains a

Write to the same variable block. Due to the cyclic nature of loops, any Write to Variable block inside the loop appears in between the Read from Variable block outside the loop and its use inside the loop.

Restart Capability

AutoCode generates code that supports application restart capability—that is, an application that is being run can be stopped and restarted again without having to download it again. Although this feature is useful, it is expensive because restarting an application requires restoring the initial data. In order to accomplish this, all the initialization data has to be stored, thus increasing the static storage size. The restart capability is useful during the development phase of an application. As the application development nears completion and is ready for deployment, the need for restarting an application might not arise. AutoCode provides an option to optimize this capability so that the optimal version does not carry this extra information.

The AutoCode command option `-Onorestart` optimizes away:

- Extra variables that store the initialization values
- Code that is used for the initialization purpose

Example 7-3 shows generated code that uses the restart capability.

Example 7-3 Sample Segment of Code with Restart Capability (Default Case)

```
void subsys_1(U, Y)
 struct _Sys_ExtIn *U;
 struct _Subsys_1_out *Y;
 {
      /***** States Array. *****/
      static struct _Subsys_1_states ss_1_states[2];
      /***** Current and Next States Pointers. *****/
      static struct _Subsys_1_states *X;
      static struct _Subsys_1_states *XD;
      static struct _Subsys_1_states *XTMP;

     static RT_INTEGER iinfo[4];
     static RT_INTEGER INIT;
     const RT_DURATION TSAMP = 0.1;
     /***** Parameters. *****/
     static RT_FLOAT R_P[8];
     RT_INTEGER cnt;
```

```
     static const RT_FLOAT _R_P[8] = {-1.0, 1.0, 1.5, 2.0, -1.0, 1.0,
1.25,1.4};

     /***** Local Block Outputs. *****/
      RT_FLOAT proc_2_1;
      RT_FLOAT proc_14_1;
      RT_FLOAT proc_12_1;

      /***** Algorithmic Local Variables. *****/
      RT_INTEGER ilower;
      RT_INTEGER iupper;
      RT_FLOAT uval;
      RT_INTEGER k;
      RT_FLOAT alpha;

      /******* Initialization. *******/
      if (SUBSYS_PREINIT[1]) {
         iinfo[0] = 0;
         iinfo[1] = 1;
         iinfo[2] = 1;
         iinfo[3] = 1;
         INIT = 1;
         X = &ss_1_states[0];
         XD = &ss_1_states[1];
         X->proc_12_S1 = 0;
         X->proc_4_S1 = 0.0;
         XD->proc_12_S1 = 0.0;
         XD->proc_4_S1 = 0.0;
         for ( cnt=0;cnt<8;cnt++ ) {
            R_P[cnt] = _R_P[cnt];
         }
         SUBSYS_PREINIT[1] = FALSE;
         return;
      }
      ..
      ..
```

Example 7-3 shows excerpts of generated code relevant to the restart capability. The buffer `_R_P` stores the initial parameter values, and during initialization time this data is copied to the `R_P` buffer, which might undergo modification. The initialization section also contains code that explicitly initializes states, state derivatives, and `info` structures. Example 7-4 shows the generated code without the restart capability. Here,

the buffer _R_P and the initialization code are optimized away. Instead, the buffer R_P (parameter array), states, derivatives, and info structures are initialized directly in the declaration portion. After these structures are modified by computations, the initial values are lost, and the application cannot be restarted again.

Example 7-4 Sample Segment of Code with Restart Capability Optimized Away

```
void subsys_1(U, Y)
struct _Sys_ExtIn *U;
struct _Subsys_1_out *Y;
{
    /***** States Array. *****/
   static struct _Subsys_1_states ss_1_states[2] = {{0.0, 0.0}, {0.0, 0.0}};

    /***** Current and Next States Pointers. *****/
    static struct _Subsys_1_states *X = &ss_1_states[0];
    static struct _Subsys_1_states *XD = &ss_1_states[1];
    static struct _Subsys_1_states *XTMP;
    static RT_INTEGER iinfo[4] = {0, 1, 1, 1};
    static RT_INTEGER INIT = 1;
    const RT_DURATION TSAMP = 0.1;

    /***** Parameters. *****/
    static RT_FLOAT R_P[8] = {-1.0, 1.0, 1.5, 2.0, -1.0, 1.0, 1.25, 1.4};

    /***** Local Block Outputs. *****/
    RT_FLOAT proc_2_1;
    RT_FLOAT proc_14_1;
    RT_FLOAT proc_12_1;

    /***** Algorithmic Local Variables. *****/
    RT_INTEGER ilower;
    RT_INTEGER iupper;
    RT_FLOAT uval;
    RT_INTEGER k;
    RT_FLOAT alpha;
    ..
    ..
```

Merging INIT Sections

Most of the dynamic blocks have explicit initialization, output update and state update sections. The initialization section is guarded by an INIT Boolean variable that is TRUE only for the first time a subsystem or a procedure is called. Each initialization section is tested every time the subsystem or procedure is executed. AutoCode supports an option where all such initialization branches can be merged into a single initialization branch.

The command option `-Oinitmerge` tries to merge all initialization segments within a subsystem or a procedure. This speeds up applications, particularly for processors with pipeline architecture such as Siemens 166/167. In this case, there would be considerable improvement in the execution speed.

If AutoCode cannot merge all the INIT sections together, it creates a separate INIT branch for the blocks that cannot be merged. This happens when a block uses outputs of other blocks executed before it is in its initialization section.

Example 7-5 shows generated code without the optimization, and Example 7-6 shows generated code with the optimization.

Example 7-5 Sample Code Segment Generated by AutoCode without this Optimization

```
/******* Initialization. *******/
if (SUBSYS_PREINIT[1]) {
  iinfo[0] = 0;
  iinfo[1] = 1;
  iinfo[2] = 1;
  iinfo[3] = 1;
  INIT = 1;
  X = &ss_1_states[0];
  XD = &ss_1_states[1];
  X->proc_22_S1 = 0.0;
  X->proc_24_S1 = 0.0;
  XD->proc_22_S1 = 0.0;
  XD->proc_24_S1 = 0.0;
  SUBSYS_PREINIT[1] = FALSE;
  return;
}

/***** Output Update. *****/
```

```
   /* ----------------------------  Time Delay */
   /* {proc..22} */
   if (INIT) {
     X->proc_22_S1 = 0.0;
   }
   proc_22_1 = X->proc_22_S1;
   /* ----------------------------  Time Delay */
   /* {proc..24} */
   if (INIT) {
     X->proc_24_S1 = 0.0;
   }
proc_24_1 = X->proc_24_S1;
/* ----------------------------  Sum of Vectors */
/* {proc..14} */
proc_14_1 = U->proc_1 - proc_22_1;
/* ----------------------------  Gain Block */
/* {proc..12} */
proc_12_1 = 2.0*proc_14_1;
/* ----------------------------  Sum of Vectors */
/* {proc..5} */
proc_5_1 = proc_12_1 - proc_24_1;
/* ----------------------------  Gain Block */
/* {proc..4} */
Y->proc_4_1 = 2.0*proc_5_1;

/***** State Update. *****/
/* ----------------------------  Time Delay */
/* {proc..22} */
XD->proc_22_S1 = proc_12_1;
/* ----------------------------  Time Delay */
/* {proc..24} */
XD->proc_24_S1 = Y->proc_4_1;
..
..
```

Example 7-6 Sample Code Segment Generated with Merging of INITs (-Oinitmerge) Option

```
 if (SUBSYS_PREINIT[1]) {
     iinfo[0] = 0;
     iinfo[1] = 1;
     iinfo[2] = 1;
     iinfo[3] = 1;

     X = &ss_1_states[0];
     XD = &ss_1_states[1];
```

```
     X->proc_22_S1 = 0.0;
     X->proc_24_S1 = 0.0;
     XD->proc_22_S1 = 0.0;
     XD->proc_24_S1 = 0.0;
     /* ----------------------------  Time Delay */
     /* {proc..22} */
     X->proc_22_S1 = 0.0;
     /* ----------------------------  Time Delay */
     /* {proc..24} */
       X->proc_24_S1 = 0.0;
       SUBSYS_PREINIT[1] = FALSE;
       return;
 }

/***** Output Update. *****/
/* ----------------------------  Time Delay */
/* {proc..22} */
proc_22_1 = X->proc_22_S1;
/* ----------------------------  Time Delay */
/* {proc..24} */
proc_24_1 = X->proc_24_S1;
/* ----------------------------  Sum of Vectors */
/* {proc..14} */
proc_14_1 = U->proc_1 - proc_22_1;
/* ----------------------------  Gain Block */
/* {proc..12} */
proc_12_1 = 2.0*proc_14_1;
/* ----------------------------  Sum of Vectors */
/* {proc..5} */
proc_5_1 = proc_12_1 - proc_24_1;
/* ----------------------------  Gain Block */
/* {proc..4} */
Y->proc_4_1 = 2.0*proc_5_1;
/***** State Update. *****/
/* ----------------------------  Time Delay */
/* {proc..22} */
XD->proc_22_S1 = proc_12_1;
/* ----------------------------  Time Delay */
/* {proc..24} */
XD->proc_24_S1 = Y->proc_4_1;
..
..
```

In Example 7-5, both time delay blocks have separate INIT sections. In Example 7-6, the initialization code for these blocks is merged along with the subsystem initialization section.

Reuse of Temporary Block Outputs

Subsystems and procedures generated by AutoCode contain computations of individual blocks. Output of these blocks could be subsystem (procedure) outputs, or it could be temporary. Temporary block outputs are used immediately in the subsequent computations. The temporary block outputs are implemented in the generated code as automatic variables and each block output is mapped by default to a unique variable. Although this improves readability and makes the generated code traceable to the model, for very large subsystems (or procedures), this could increase the stack size. AutoCode provides an option to reuse such temporary block outputs, thus reducing the stack size and bringing down the risk of stack overflow.

AutoCode provides two ways of reusing temporary block output:

- Reuse whenever specified (by the user)
- Maximal reuse (reuse whenever possible)

Reuse Temporaries as Specified

In this mode, a temporary variable is reused only if you specify it and if it is safe to do so. You can specify reuse by entering a variable name in the output name field in the output form of a block. Refer to the *SystemBuild User Guide* for more details on output forms. Even if this temporary variable was already used as an output name for another block output, AutoCode tries to reuse it. If the temporary variable is not used (or needed) anymore, then AutoCode uses it as the output variable for the current block. In a case where the variable identified cannot be reused, a distinct (entirely new) variable is used. The command-line option for this optimization mode is `-Oreuse 1`.

Maximal Reuse of Temporaries

In this optimization mode, AutoCode tries to reuse as many temporary variables as it can. Again, a temporary variable is reused only if it is not needed in any further computations. The command-line option `-Oreuse 2` brings about this optimization.

Example 7-7 shows code generated without the reuse option, and Example 7-8 shows code generated from the same models with the maximal reuse option.

Example 7-7 Code Fragment without Reuse Optimization

```
/***** Local Block Outputs. *****/
  RT_FLOAT model_2_1;
  RT_FLOAT model_3_1;
  RT_FLOAT model_4_1;

/***** Output Update. *****/
/* ----------------------------   Time Delay */
/* {model..5} */
if (INIT) {
  X->model_5_S1 = 0.0;
}
Y->model_5_1 = X->model_5_S1;

/* ----------------------------   Gain Block */
/* {model..2} */
 model_2_1 = 2.0*U->model_1;
 /* ----------------------------  Sum of Vectors */
 /* {model..3} */
 model_3_1 = model_2_1 - U->model_1;
 /* ----------------------------  Gain Block */
 /* {model..4} */
 model_4_1 = 3.0*model_3_1;
 /***** State Update. *****/
 /* ----------------------------  Time Delay */
 /* {model..5} */
 XD->model_5_S1 = model_4_1;
```

Example 7-8 Code Fragment Generated with Maximal Reuse Option (-Oreuse 2)

```
 /***** Local Block Outputs. *****/
 RT_FLOAT model_2_1;
 RT_FLOAT model_3_1;

/***** Output Update. *****/
/* ----------------------------  Time Delay */
/* {model..5} */
if (INIT) {
  X->model_5_S1 = 0.0;
```

```
}
Y->model_5_1 = X->model_5_S1;
/* ----------------------------  Gain Block */
/* {model..2} */
model_2_1 = 2.0*U->model_1;
/* ----------------------------  Sum of Vectors */
/* {model..3} */
model_3_1 = model_2_1 - U->model_1;
/* ----------------------------  Gain Block */
/* {model..4} */
model_2_1 = 3.0*model_3_1;
/***** State Update. *****/
/* ----------------------------  Time Delay */
/* {model..5} */
XD->model_5_S1 = model_2_1;
```

In Example 7-7 (code generated without reuse option), each block has a distinct and unique output variable. In Example 7-8 (code generated with the maximal reuse option), the variable `model_2_1`, which is the output variable for the block `model..2`, is used only in the computation for block `model..3`. It is free to be used again after the computation for `model..3` is complete, and in fact the generated code reuses the variable `model_2_1` as the output variable of the block `model..4`.

Constant Propagation

AutoCode supports an option to propagate constants in the generated code. Source of constants in a model are typically algebraic and logical expression blocks. For the sake of this optimization, you can partition all of the blocks into two categories.

- Blocks that can propagate constants
- Blocks that cannot propagate constants

Most of the blocks from palettes such as ALG, PWL, LOG, TRG, and PEL, belong to the first category (propagate constants), and if all of the inputs to such blocks are constants, the output value is computed during compile time, and no code is generated for such a block. The blocks belonging to the second category (cannot propagate constants), are from other palettes such as DYN, SGN, and NTP. These blocks do not propagate outputs even if all of their inputs are constants. Code is generated for such blocks.

All of the blocks can accept constants as inputs. If any input is available during code generation time as a constant, the constant is used instead of a symbol name (or variable name). This applies to blocks of both categories.

One limitation to this optimization is that a block cannot propagate constants or even accept constants as inputs if its inputs or outputs contain vectors. Such blocks with vector inputs or outputs use only the variable name (or symbol name) and hence force the source blocks to generate code for computing the value of output variables.

Example 7-9 shows code generated without constant propagation, and Example 7-10 shows code generated with constant propagation.

Example 7-9 Code Generated without the Constant Propagation Option

```
void subsys_1(Y)
struct _Subsys_1_out *Y;
{
   static RT_INTEGER iinfo[4];
   /***** Local Block Outputs. *****/
      RT_FLOAT const_1_1;
      RT_FLOAT const_11_1;
      RT_FLOAT const_2_1;

      /******* Initialization. *******/
      if (SUBSYS_PREINIT[1]) {
        iinfo[0] = 0;
        iinfo[1] = 1;
        iinfo[2] = 1;
        iinfo[3] = 1;
        SUBSYS_PREINIT[1] = FALSE;
       return;
}

/***** Output Update. *****/
/* ---------------------------  Algebraic Expression */
/* {const..1} */
const_1_1 = 5.0;
/* ---------------------------  Algebraic Expression */
/* {const..11} */
const_11_1 = 3.0;
/* ---------------------------  Sum of Vectors */
/* {const..2} */
const_2_1 = const_1_1 - const_11_1;
/* ---------------------------  Gain Block */
```

```
/* {const..4} */
Y->const_4_1 = 2.0*const_2_1;
..
..
```

Example 7-10 Code Generated with the Constant Propagation Option

```
void subsys_1(Y)
struct _Subsys_1_out *Y;
{
   static RT_INTEGER iinfo[4];
   /***** Local Block Outputs. *****/

  /******* Initialization. *******/
  if (SUBSYS_PREINIT[1]) {
    iinfo[0] = 0;
    iinfo[1] = 1;
    iinfo[2] = 1;
    iinfo[3] = 1;
    SUBSYS_PREINIT[1] = FALSE;
    return;
}

/***** Output Update. *****/
/* ---------------------------  Algebraic Expression */
/* {const..1} */

/* ---------------------------  Algebraic Expression */
/* {const..11} */

/* ---------------------------  Sum of Vectors */
/* {const..2} */

/* ---------------------------  Gain Block */
/* {const..4} */
Y->const_4_1 = 4.0;
..
..
```

In Example 7-10, during code generation time, AutoCode figures out that the outputs of both the algebraic expressions are constants and feeds them to the summation block. As all of the inputs to the summation block are constants (5 and 3), AutoCode computes the output of the summation block at code generation time (`const_2_1 = 5 - 3 = 2`). This becomes the

input of the gain block and the expression `2.0 * const_2` is evaluated to 4 as the value of `const_2_1` is 2. Hence, the subsystem output gets the value 4.

The command-line option for invoking this optimization is `-Opc`.

Optimizing with Matrix Blocks

Outputs labeled as matrices are generated as vectors, so most matrix blocks can be optimized by following the rules you would follow for any other vectorized block:

- Try to group all the outputs into a single vector (or matrix) so the output variable will be an array
- Try to connect all the inputs from the same output array of a single block, in a smooth stride

In general, this leads to looped code rather than unrolled code being generated for a block—which reduces the code footprint—or at least to the fewest number of loops per block. This applies to the MatrixTranspose, MatrixMultiply, LeftMultiply, RightMultiply, and ScalarGain blocks.

Optimizing with Constant Blocks

Constant blocks are highly optimized by design. When you create a constant block and label the output as a single matrix, you will not see any code generated for the block—the constant values defined in the block will be part of the declaration of the block output variable at the top of the containing subsystem or procedure.

By default, such output variables are declared as type “static” in C or “constant” in Ada and thus should not require separate assembly instructions to load values into position. For C, you can make them stack variables (automatics) by modifying the definition of `RT_CONSTANT` in the `sa_types.h` file. If you declare the output variable to be a stack variable, assembly instructions are generated to load its values into place each time the containing subsystem or procedure is executed.

The previous information applies provided you have not used a %Variable in the block. If a %Variable is present, a different optimization takes over. Instead of putting values into the declaration—which you cannot do in the case of a %Variable—you perform forward propagation on the constant block, replacing references to the constant block with direct references to the associated %Variable. If all use points can be so optimized, the entire

constant block is optimized away, including the output variable. Also notice that the existing Constant Propagation optimization can be used with the constant block, but will only operate on scalar pieces of the constant block output.

Optimizing with Callout Blocks

The MatrixInverse, MatrixRightDivide, and MatrixLeftDivide blocks are implemented with callouts and therefore carry a special set of rules that must be followed to generate optimal code. The most important rule, applicable to all three blocks, is that the block output should be labeled as a single matrix or vector (or generate code using maximal vectorization). This is because the algorithms associated with these blocks are stand-alone entities with fixed interfaces; a single array is used for the block output for the MatrixInverse, MatrixRightDivide, and MatrixLeftDivide algorithms.

When the output of a callout block in the generated code is spread among several different variables, a copy-back must be emitted after the callout to ensure the results are correctly stored in the desired output variables. Following this rule can have a significant impact on code generation for these three blocks. However, regarding input rules, the callout interface introduces certain constraints on inputs if optimality is desired.

Optimizing with Inverse Blocks

The MatrixInverse block callout has a single argument which is both the input and output to the inversion algorithm. Thus, the input is modified by the algorithm. For this reason, a copy-in must always be done for this block and the input connectivity is not nearly as important as the output connectivity. You may see tighter code with good input connectivity, because the copy-in will be looped rather than unrolled, but the copy-in will still be present.

Optimizing with Division Blocks

The two division blocks: MatrixRightDivide and MatrixLeftDivide, solve the equations XA = B and AX = B, respectively. When you consider which to use for your application, notice that you will get more efficient code by using the MatrixRightDivide rather than the MatrixLeftDivide. This is because AutoCode generates output matrices in row-major order, whereas the LINPACK callouts are written expecting column-major inputs. Solving AX = B under such mismatched conditions requires an extra transpose-copy not required to solve XA = B.

If you have decided on the MatrixRightDivide block, the tips for optimizing the inputs are much the same as for the MatrixInverse block. In this case there are two inputs, A and B, and both are modified by the callout algorithm. Thus, a copy-in must occur for each input. Again, you cannot avoid the copy-in, but you can get tighter code with good input connectivity because the code will be looped rather than unrolled.

If you cannot avoid using the MatrixLeftDivide block, you should know how to cause it to be generated with maximum optimality. This block has a built-in optimization that kicks in when the B matrix (from AX=B) is a row or column vector. When this built-in optimization is active, both the A and B inputs are modified by the callout algorithm and each must be subjected to a copy-in. Good input connectivity will let you generate looped rather than unrolled code, but the copy-in cannot be avoided. Furthermore, the disadvantage to using the MatrixLeftDivide block does not apply with the optimization active.

If your B matrix is not a row or column vector, then the algorithm must do an internal transpose-copy. Under these circumstances, the A input is modified by the algorithm, but the B input is not. Thus, the copy-in is unavoidable for the A input but will only be present for the B input if you have bad connectivity.

For this reason, it is very important when dealing with a MatrixLeftDivide that has a true matrix (not $1 \times n$ or $n \times 1$) as B that you have good input connectivity—just as important as the rule mentioned earlier about applying to all three callout blocks encouraging good output connectivity. In this case, good input connectivity for B means that B can be taken in place from the source matrix.

For a summary of inputs and outputs for callout blocks, refer to Table 7-1.

Table 7-1. Optimization Table for Callout Blocks

Block	Output (X)	Input A	Input B
MatrixInverse	Copy only if output is not a single array	Mandatory copy-in	Not applicable
MatrixRightDivide or MatrixLeftDivide (if B is z × n or n × 1)	Copy only if output is not a single array	Mandatory copy-in	Mandatory copy-in
MatrixLeftDivide (if B is m × n, m > 1, or n > 1)	Copy only if output is not a single array	Mandatory copy-in	Copy only if input connectivity prohibits using source array directly

Summary

All of the optimizations discussed so far can be used at the same time without limitations. All of these optimizations can work together and potentially bring about a cumulative effect in reducing the code size or improving the execution speed of an application. In general, the restart optimization, varblock read, and constant propagation can help reduce the code size, while merging of INITs, constant propagation, and varblock read optimizations improve execution speed. The reuse of temporaries helps reduce stack size, and could prevent stack overflow problems. Prudent use of the Constant Block can reduce code size, and its use is preferred over the Algebraic Expression Block for defining constants. By using these optimizations, you can generate optimal C or Ada source code. The executable (or the object) file size depends on the compiler used for compiling the source code. Execution speed of an application ultimately depends on the target processor.

8

AutoCode Sim Cdelay Scheduler

This chapter discusses the Sim Cdelay low-latency scheduler.

Introduction

The default AutoCode scheduler is based on high throughput. Latency is acceptable as long as scheduler interrupt times are frequent. Because of this emphasis, each scheduler cycle is kept to an absolute minimum so that task queuing and output posting occurs only once per cycle. While providing high throughput and determinacy, this strategy has an impact on the latency of triggered and enabled tasks. For example, after an enable goes high, it takes one cycle for the task to be queued, and a second cycle before the output gets posted. This process works similarly for triggered tasks. The upshot is that the current AutoCode scheduler incurs a two-cycle delay when firing off any enabled or triggered tasks, and a sequence of such delays in firing off chains of such tasks.

The MATRIXx product line lets you reproduce the behavior of the generated code with Sim. To match the default AutoCode scheduler, Sim can be invoked with the `actiming` option. However, a Sim user also has available two simulation options that do not mimic the default AutoCode scheduler:

- Sim with Cdelay
- Sim without Cdelay

While Sim without Cdelay represents an unrealistic goal for a real-time system (no computational delay for any device), Sim with Cdelay not only is realizable on real-time hardware, it avoids the latency problems that plague Sim with `actiming`, while preserving determinacy. Its only drawbacks are that it requires a slightly longer overhead per scheduler invocation, and it cannot completely escape the latency problems present in the default AutoCode scheduler if chained ANC tasks are present in the model. Despite these limitations, it is vastly superior to the default AutoCode scheduler at latency reduction.

The remainder of this chapter describes a new AutoCode scheduler that matches the Sim with Cdelay behavior and is capable of executing on

real-time hardware. It assumes the reader is familiar with the concepts of AutoCode's default scheduler. Refer to the *AutoCode User Guide* for more information about the scheduler, task types, and output posting.

Task Posting Policies

The task characteristics that have the most impact on scheduler design are task type, and output posting policy. For real-time applications, there are two workable, mutually exclusive posting strategies a task can adopt:

- ANC, or At Next Computation
- ATR, or At Timing Requirement

A task, which is ANC, is invoked, starts its computation and, after the result has been calculated, caches it without posting. The next time that task is invoked, the cached result is posted before the computation starts. The important point about ANC tasks is that enabling or triggering them can generate an immediate output, even in real-time applications, without waiting for a computational delay.[1] Conversely, when an ATR task is invoked, it starts its computation, caches the result, and posts it exactly *t* minimum scheduler cycles after its invocation, where *t* is the timing requirement.

Now that the posting policies are understood, the map from task types to posting policies can be studied. For free-running periodic tasks, you do not even need the concept of a posting policy, because the difference between posting at the end of one cycle and posting at the beginning of the next (before inputs are sampled and held) is meaningless. For triggered tasks, the posting policy is necessary and sufficient to provide a description; ANT triggered tasks are ANC, while ATR/SAF triggered tasks are ATR. For SAF triggered tasks, the timing requirement is assumed to be the minimum scheduler cycle. Finally, notice that from what is stated above the output posting policy of enabled periodic tasks is not yet determined—it can be either ANC or ATR.

With enabled tasks, there is also a notion of the enable policy, which determines when the task is invoked rather than when it posts its outputs. You can demand that enabled tasks are launched only on the global timeline major cycle points for tasks of that rate, or allow such tasks to be immediately launched on the first scheduler minor cycle that the enable is known to be high.

[1] Later in this chapter, this will be shown to lead to the ANC Chaining Problem.

Standard AutoCode Scheduler

To illustrate the behavior of the standard AutoCode scheduler with triggered and enabled SuperBlocks, it is helpful to consider the diagnostic model shown in Figure 8-1, which features such SuperBlocks driven by a 5 Hz pulse train and fed a time signal as input.

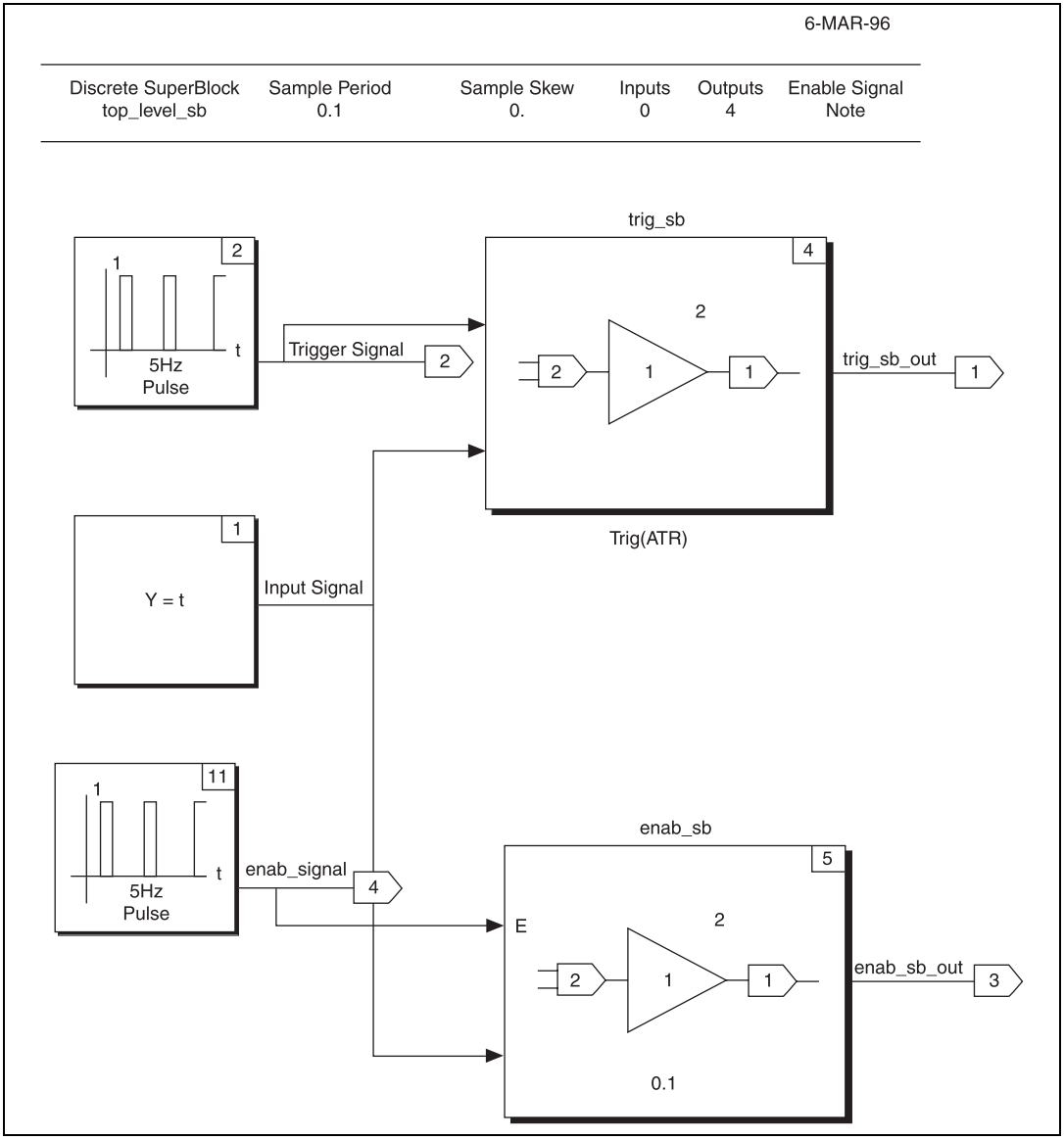


Figure 8-1. Model with Enabled and Triggered Tasks

Under the default AutoCode scheduler, the output of this system is as shown in Figure 8-2. The top signal (corresponding to the triggered task) shows a latency of two cycles relative to its trigger input (immediately below), while the lower signal (corresponding to the enabled task) shows a latency of three cycles relative to its enable input. You will observe in the

Scheduler Pipeline section that these latencies are caused by the single posting stage per scheduler invocation in the default AutoCode scheduler, and by the output posting policy assigned to enabled blocks in the default scheduler.

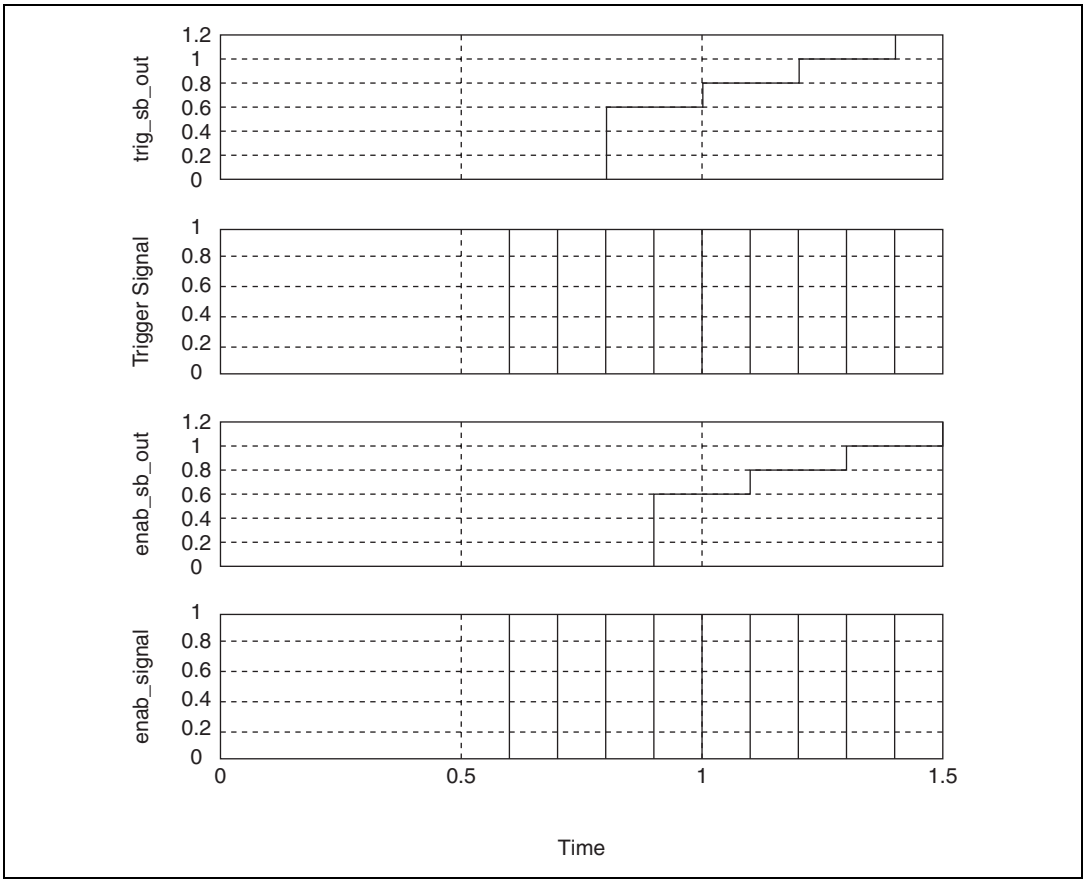


Figure 8-2. Latencies Present in Default AutoCode Scheduler

Scheduler Pipeline

To understand these latencies, you need to know what operations are performed during each scheduler invocation under the default scheduler. These are best presented in the form of a pipeline diagram as shown in Figure 8-3. Stage A resets the priority associated with each DataStore register so that anyone can write to it. Stage B is where any interpolation occurs to generate internally needed timepoints from an irregularly-spaced input vector. In stage C, tasks whose launch times have arrived or whose

triggers/enables have gone high are queued for execution. Then, in stage D, ATR tasks whose output posting times have arrived and ANC tasks that have just been queued for execution in stage C are marked for output posting, which is stage E.[1, 2] Datastores are written from external inputs in stage F (overwriting any previous values for that cycle). Finally, in stage G, queued tasks are dispatched for execution.

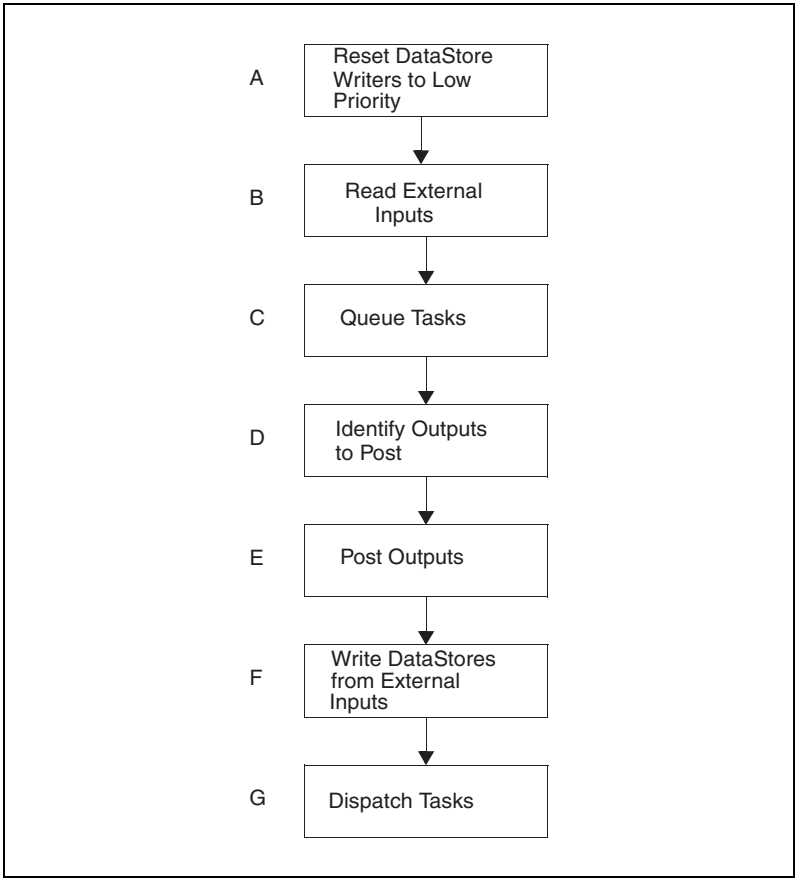


Figure 8-3. Scheduler Pipeline Stages in the Standard AutoCode Scheduler

Given this flow, you now can explain why the delays are present for the triggered and enabled subsystems. Take the case of the triggered subsystem first. When the triggered task's trigger input is posted high at 0.6 sec, the

[1] Notice that in the standard template, certain loops internal to a stage have been lifted to include two or three stages—for example, stages D, E, and F.

[2] Any DataStores driven by task outputs are written to during this cycle.

queue tasks stage for that scheduler invocation has already passed; thus, the triggered task cannot be queued for execution until 0.7 sec. At the time that it is queued, it sets its time delay for 0.1 sec. Thus, its output is not posted until the next scheduler invocation. In total, two scheduler ticks have thus passed before the triggered task's outputs have been posted.

The case of the enabled task is slightly more complicated. In this case, you care not only about the scheduler pipeline itself but also about the output posting policy of the task in question. Enabled tasks under the default AutoCode scheduler are assigned a post on next computation output paradigm. Now consider the output in Figure 8-1 for the enabled task; the enable input rises at 0.6 sec. As this corresponds to the post outputs stage of a scheduler invocation, the queue tasks stage has already occurred. Therefore, you must wait until 0.7 sec, when you enter the next scheduler invocation, for the enabled task to be queued for execution.

Now you must bring into play the fact that enabled periodic tasks are ANC in the default scheduler; results of an invocation of an enabled task are only posted on the next invocation of that task. At the end of the 0.7 sec scheduler invocation, the enable input goes low, which means that the enabled task is not enabled for the 0.8 sec scheduler invocation. Thus, the task is not invoked again, and it posts no output; however, its enable input does go high at the end of this cycle. Finally, at the 0.9 sec scheduler invocation, the enabled task is invoked again and the cached result from the last computation is posted. The total observed delay for the enabled task, from enable going high to output being posted, is thus three scheduler ticks. In the case of this and the preceding analysis, it might benefit you to refer to the state transition diagrams (default scheduler) for each type of task in Chapter 4, *Managing and Scheduling Applications*, of the *AutoCode User Guide*.

Managing DataStores in the Scheduler

Notice that DataStores in a model do not translate into tasks—they are handled separately by the scheduler during the output posting phase of the scheduler pipeline. As there could be multiple attempts to write into a DataStore during a single scheduler invocation, a constraint that must always be observed is that a task can write into a DataStore on a particular scheduler cycle if and only if no task of higher priority has already written to it. In the default scheduler, this is trivially enforced by ordering the firing of tasks in the (single) output posting stage so that lower priority tasks fire first, but the same constraint will become of key importance (and a potential stumbling point) when it comes time to implement the new scheduler.

Now the inherent problem in the standard scheduler is clear. From launch to output posting, you suffer a two-cycle delay for triggered tasks, and a three-cycle delay for enabled tasks, instead of the standard unit cycle delay present on real-time hardware for free-running periodic tasks. Your goal will be to reduce both of these latencies to single-cycle delays. The scheduler pipeline stages in the Sim Cdelay AutoCode scheduler are shown in Figure 8-4.

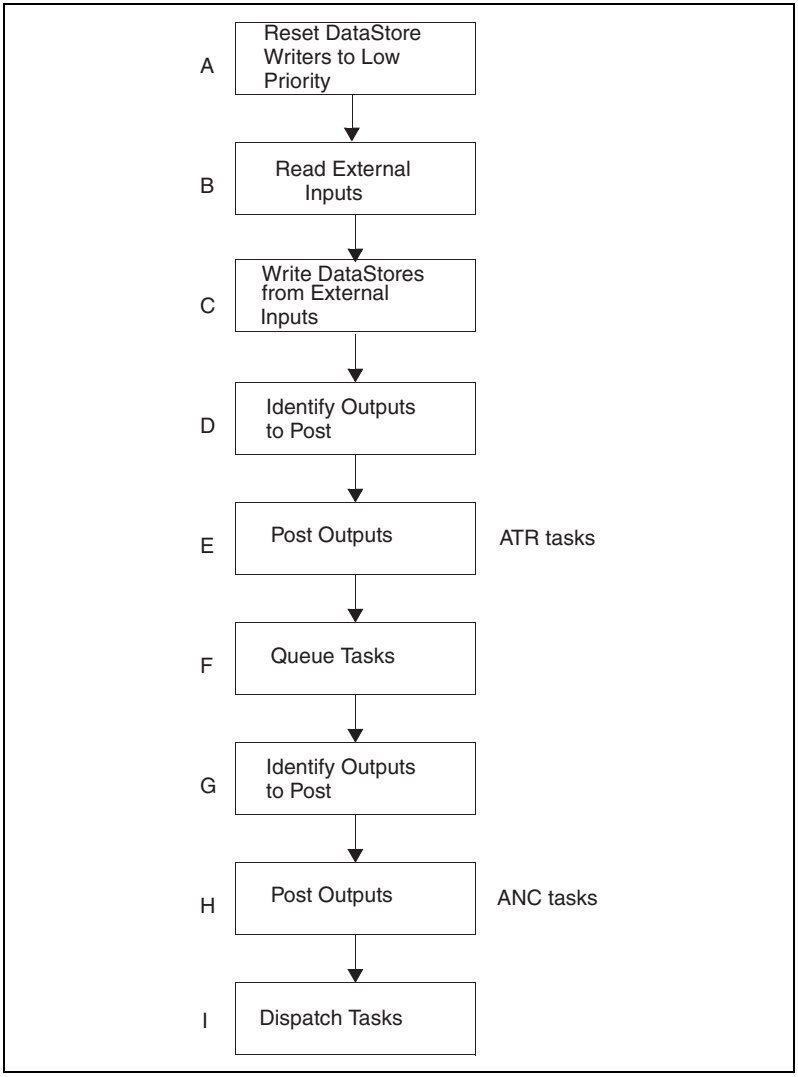


Figure 8-4. Scheduler Pipeline Stages in the Sim Cdelay AutoCode Scheduler

Sim Cdelay Scheduler

As stated at the outset, the goal of this project was to develop a new AutoCode scheduler, runnable on real-time targets, which mimics the behavior of Sim with Cdelay (`actiming` off). Sim with Cdelay differs from Sim with `actiming` in several key ways, including scheduler pipeline, task output posting policies, enable policies, and retriggering behavior for triggered tasks.

Notice that, unlike the stock AutoCode scheduler, Sim with Cdelay's scheduler posts outputs twice per invocation. Outputs of tasks that are ATR are posted exclusively during the first output posting stage, while tasks that are ANC have their outputs posted during the second output stage. Between the two output postings stages is the queue tasks phase, which queues up those tasks that are ready to start executing that cycle. Because all the ATR tasks post their outputs before this stage, it is clear that under this new pipeline configuration, any task enabled or triggered by an ATR task will be queued for execution during the same cycle its input goes high.

In addition to the altered scheduler pipeline, Sim with Cdelay also entails new output posting options for certain tasks. In fact, the output posting policy of all periodic blocks (both free-running and enabled) goes from ANC (Sim with `actiming`) to ATR (Sim with Cdelay). For free-running tasks, ANC versus ATR does not really make a difference—it is just a paradigm shift. However, for enabled periodic tasks, the shift from ANC to ATR leads to a very tangible reduction in output posting latency.

One of the remaining changes brought by Sim with Cdelay is the move from a sync immediate on enable policy (default AutoCode scheduler), in which the enabled task is queued for execution on the first minor cycle its enable input is seen to go high, to a global timeline enable policy. Unlike the previous two modifications, this switch has a deleterious effect on output latency. The enable policy only matters in a given model if the scheduler minor cycle differs from the major cycle of one of the tasks; thus, it has no effect on the model presented at the beginning of this chapter.

The other change entailed by Sim with Cdelay affects how retriggering is handled for ATR triggered tasks if the computation has concluded but the task has not yet posted its outputs. The default scheduler queues the trigger (at most one), releasing it (and allowing the system to be triggered again) only when its outputs are finally posted. Conversely, with Sim with Cdelay, the trigger is immediately acted on, even though the outputs have not been posted. Notice that this has the unfortunate consequence that a sequence of

triggers arriving too quickly can prevent a task from ever posting any output.

You must build an alternative AutoCode scheduler incorporating the new pipeline stages presented above, and change the transition diagrams of many of the task types to reflect changed output posting, enable, and retriggering policies. In the case of the enable and retriggering policies, however, you will want to include switches that let you fall back to the original AutoCode behavior for these options because the Sim choices have the previously mentioned drawbacks.

State Transition Diagrams of Tasks under Sim Cdelay

New transition diagrams were developed for free-running and enabled periodic tasks, and for ATR triggered tasks. Figure 8-5 shows a new STD for free-running periodic tasks. Figure 8-6 shows a new STD for ATR triggered tasks. Figure 8-7 shows a new STD for ATR triggered tasks.

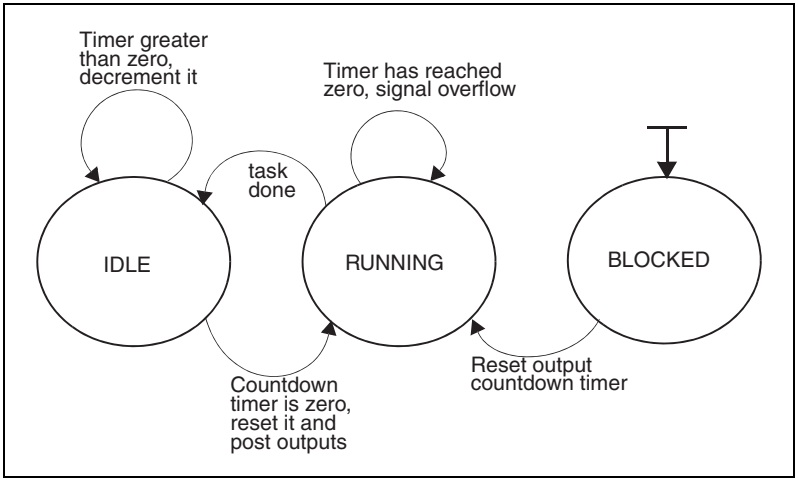


Figure 8-5. New STD for Free-Running Periodic Tasks

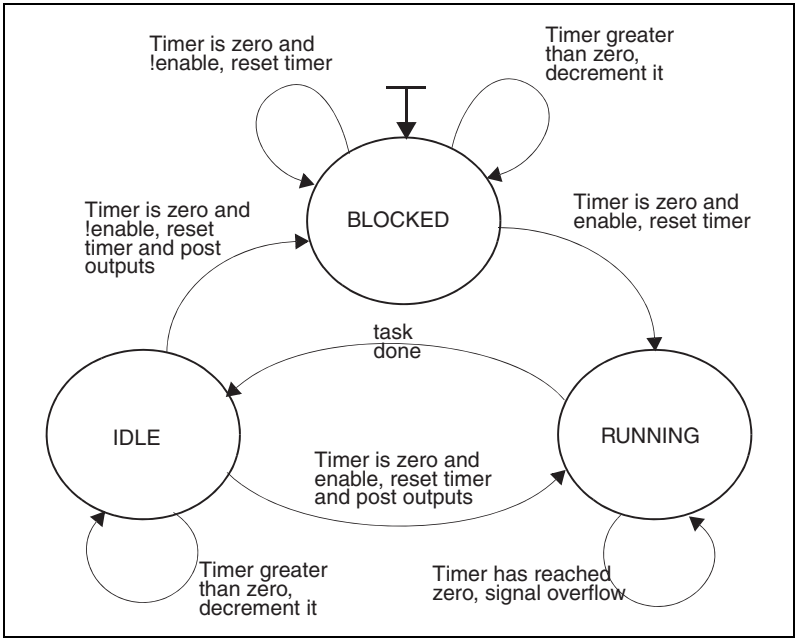


Figure 8-6. New STD for ATR Triggered Tasks

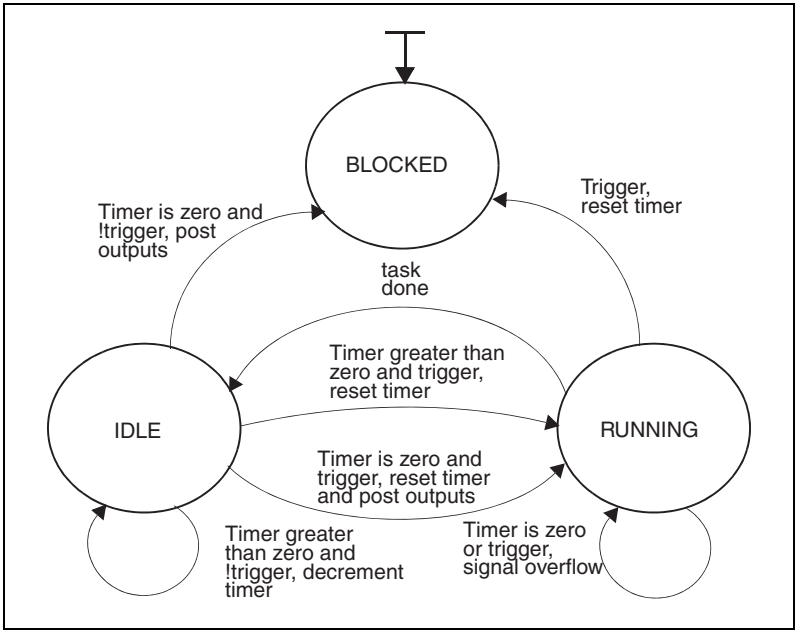


Figure 8-7. New STD for ATR Triggered Tasks

These transition diagrams, together with the diagrams in Chapter 4, *Managing and Scheduling Applications*, of the *AutoCode User Guide*, define the behavior of tasks under the Sim with Cdelay scheduler. They encompass all of the changes to the default scheduler outside the scheduler pipeline, and embody the new output posting, enable, and retriggering policies. Together with the new pipeline configuration, these diagrams provide you with enough information to completely implement the new scheduler.

Implementing the Sim Cdelay AutoCode Scheduler

The new scheduler can largely be implemented as a new template. In the new template, the large-scale layout of the scheduler is determined by the new pipeline configuration, and the variable manipulations inside each scheduler stage are determined by the STDs for the various task types.

Implementation Details

Some of the implementation details are as follows:

- When a task attempts to write to a DataStore, the write must only be allowed to go through if no task of higher priority has written to that same location on the current scheduler cycle. In the default AutoCode scheduler, this is enforced by ordering the posting of outputs to post from least priority task first and the highest priority task last, thereby obviating the need for active checks. With the new scheduler, however, you have two separate output posting stages. As a result, it is necessary to develop an active protection mechanism whereby the priority of a writer to a DataStore register is stored, and later used in a guard. This will be discussed in depth in the *DataStore Priority Problem* section.
- The output posting policy for all periodic tasks changes from ANC to ATR in the transition to the new scheduler. Thereby, the periods of periodic tasks should be stored in the template parameter `OUTPUTCOUNT_LI`. In the default scheduler, this vector is zero except for ATR triggered tasks. The timing requirements also can be obtained from `SCHEDULINGCOUNT_LI`, as they are for the (ANC) periodic tasks in the default scheduler, but this is not good design.
- Under the new scheduler, free-running periodic tasks have a blocked state used only during start-up so they can be initialized to have timer values of zero without causing meaningless outputs to be posted. Under the default scheduler, there is no blocked state for free-running periodic tasks; thus, the template parameter `INITIALTASKSTATE_LS` must be altered.

DataStore Priority Problem

As mentioned, you must find some way to enforce the priority of writers to DataStores in the generated code. Each DataStore register can be written to independently, so each one needs to be independently subjected to the priority constraint. In the default scheduler, there is no need to actively enforce the constraint, as it is automatically preserved by the order in which outputs are posted in the single output stage. However, in the new scheduler, there are two output posting stages, and the same technique will not work.

An efficient solution—the one implemented—involves noticing that only the registers in DataStores written to by both ANC and ATR tasks need be actively monitored. For example, if a register is written to only by ANC tasks, it is written entirely in the first stage and ordering the writes from lowest to highest priority would enforce the constraint as in the default scheduler. So you only need to create a priority array with just enough space to hold priorities of registers in DataStores written to by both types of tasks. Furthermore, observe that the problem is caused by the need to preserve information across the gap between the first and second output posting stages. You do not care about the regime before the first stage or after the second.

Thus, if you call the set of registers in DataStores written to by both ANC and ATR tasks Alpha, it would suffice to record the priority of each writer to a member of Alpha only in the first output stage, because within the second stage repeated writes do not matter because of the order in which the write loop is executed. Likewise, guards are not needed in the first output stage because of the write loop order. All you need to do is record the priority of the writers to members of Alpha in the first stage, and wrap writes to members of Alpha with guards in the second stage.

Some pseudocode for this, assuming a single DataStore register in Alpha, is shown in Example 8-1.

Example 8-1 Pseudocode for Single DataStore Register in Alpha

```
for tsk <- low_prio_tsk..high_prio_tsk do {
            /* loop from low to high priority */
if tsk.ready_to_post_outputs() {
   write_register(tsk.output());  /*write to datastore register*/
   reg_prio = tsk.prio();         /*save priority of writer*/
   }
}
/* Queue Tasks */
```

```
for tsk <- low_prio_tsk..high_prio_tsk do {
           /* loop from low to high priority */

if tsk.ready_to_post_outputs()
  if tsk.prio() > reg_prio /* higher priorities are larger */
    write_register(tsk.output()); /*write to datastore register*/
```

In the generated code, the priorities of all DataStore registers in Alpha are stored in a single, file-scoped array called `DSTORE_GUARD`, which is optimized away if Alpha is empty.

Using the Sim Cdelay Scheduler

With the new template in place, the choice of scheduler can be made easily at AutoCode run time by use of a new command-line option. Given a current version of AutoCode and the new template, using it without any new command options generates code with the default scheduler algorithm. Using it with a `-sched 1` option, however, switches in the new scheduler. Issuing the command:

```
% autostar -l c -sched 1 trig_model.rtf
```

should, for example, generate code in `trig_model.c` employing the new scheduler. Under the Sim with Cdelay scheduler, the model of Figure 8-1, behaves as shown in Figure 8-8. The latency of the triggered and enabled SuperBlocks has been reduced to a single scheduler tick.

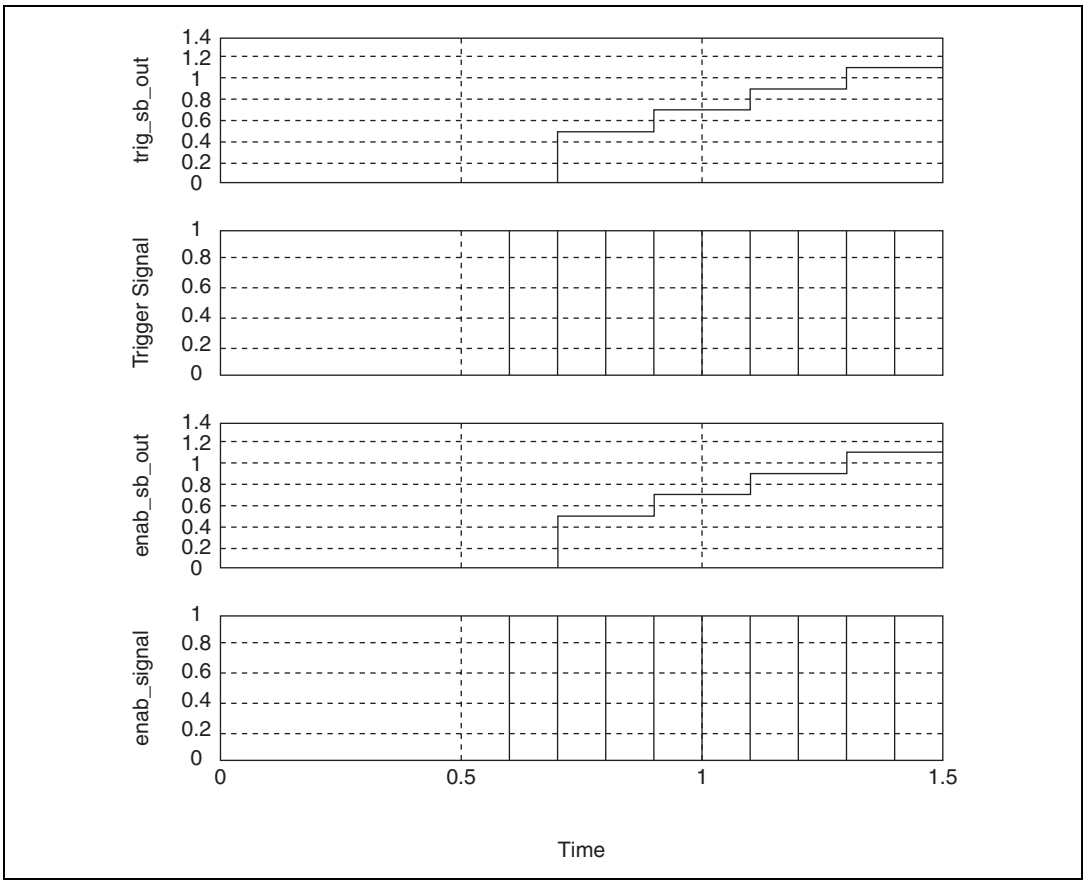


Figure 8-8. Latencies Present in the AutoCode Scheduler

Template Configuration for Enhanced Performance

As mentioned in a previous section, there are drawbacks to re-executing an ATR task when it is re-triggered before its outputs have been posted, and to adopting a global timeline enable policy (increased latency). Thus, a transition diagram is provided in Figure 8-9 can be substituted for that of Figure 8-6 to obtain the main features of Sim with Cdelay, while retaining the original AutoCode enable policy (sync immediate on enable). The original AutoCode retriggering policy can be recaptured by using the STD for ATR tasks shown in Chapter 4, *Managing and Scheduling Applications*, of the *AutoCode User Guide*, in lieu of Figure 8-7.

At the template level, the default Sim with Cdelay ATR triggered task behavior of re-queueing a task for execution on receipt of a new trigger before the outputs have been posted can be turned off by replacing the segment call `Sim_style_ATR_Idle` by a call to `NoRetrigger_style_ATR_Idle`. To turn off the Sim with Cdelay global timeline enable policy, replacing it with the default sync immediate on enable policy, you must replace the calls to the segments `Sim_style_Enable_Idle` and `Sim_style_Enable_Blocked` with calls to the segments `Fast_style_Enable_Idle` and `Fast_style_Enable_Blocked`, respectively.

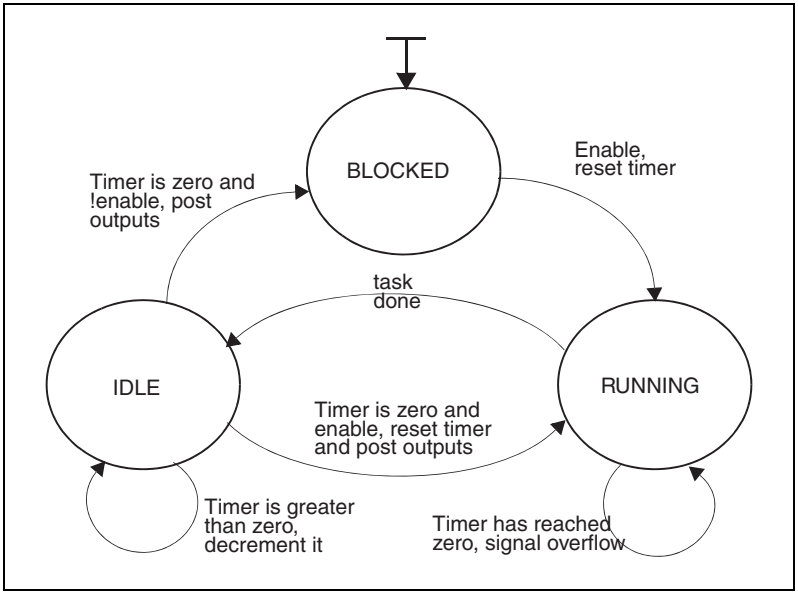


Figure 8-9. Alternative (Old Enable Policy) STD for an Enabled Task

Shortcomings of the Sim Cdelay Scheduler

What the pipeline re-configuration did in the new scheduler was to purchase low latency at the price of scheduler overhead. However, the solution is not complete; given a chain of ANC tasks—that is, the output of an ANT triggered block is the trigger of another ANT triggered block—the new scheduler would not fire off all elements of the chain in a single cycle as an ideal scheduler would. Instead, it would only catch the first element of the chain; each additional element would suffer a cycle of scheduler latency. To solve this problem, it would suffice to enclose the queue task and the second post outputs stage of the new scheduler in a relaxation loop

that repeated until no new tasks where queued for execution. Given such an implementation, the chained ANC problem would disappear. However, under such a design, scheduler execution time would be variable and data dependent—undesirable attributes in any hard real-time scheduler. Contrast this with the new scheduler, which catches the first element of an ANC chain but has a fixed-length pipeline. An evaluation of the trade-offs between the relaxation approach and the fixed pipeline approach must be done before such potentially dangerous changes are made to any scheduler.

9

Global Scope Signals and Parameterless Procedures

This chapter discusses global scope signals and parameterless procedures.

Introduction

The memory and performance requirements of real-time production code force the issue of global variables. AutoCode does not generally use global variables; rather, it creates and uses stack variables and explicit interfaces. Whereas that architecture is perfectly sound, it does cause overhead in a production system. Therefore, AutoCode now supports direct use of global variables for local block outputs within a subsystem. Extending that concept allows global variable(s) to be used as the inputs and outputs of a procedure.

You must explicitly select which signals within the model will be generated as global variables. In addition, you also will have to specify which of the procedure inputs and outputs are to be global. Fortunately, you are able to specify these signals within the SuperBlock Editor. A summary of the capabilities regarding these global signals includes:

- Model-level specification of global signals
- Specification of a memory address that can be attached to each global signal
- Specification of which procedure inputs and/or outputs are to be global variables
- Verification of connectivity during Analyze-time
- Access to global variable names from within the template, allowing customized locality of the declarations
- Command-line override to force all signals to be global or not global
- DocumentIt support

These new features address the following two uses:

- **Data Monitoring/Injection**—The safe access to local subsystem signals for the purpose of monitoring the signal's value during the execution of the model. This is intended as a way to provide debugging/monitoring information.
- **Parameterless Procedure**—The ability to use global variable(s) to represent a procedure's inputs and/or outputs. This involves a clear specification of the names of those global variables for both the procedure and the caller.

Data Monitoring/Injection

Data monitoring is the ability to have read-only access signals within a subsystem from outside the subsystem by some external agent outside the scope of the SystemBuild model. Data injection is the ability to modify signals within a subsystem from outside the subsystem by some external agent outside of the SystemBuild model. This is usually performed in a testing environment to determine the effect of extraordinary events on the system being tested. Both data monitoring and data injection require that the automatic variables that are normally used for local blocks outputs instead be global variables. When the block outputs are global variables, the values persist after the subsystem exits for that time-cycle. Because the AutoCode requirements for monitoring and injection are identical, the remainder of this section will discuss the monitoring case.

You can monitor almost every signal from a basic block within a subsystem. AutoCode will ensure that the naming of these global variables used for monitoring are unique across the whole model. Any conflicts will be automatically resolved by mangling the variable name.

Specifying Monitored Signals

A monitored signal is generated as a safe global variable. A safe global variable will generate as a unique variable, regardless of the signal's name/label and whether it matches another block's signal name/label.

Within the SuperBlock Editor, you have the choice of specifying which outputs of a basic block will be monitored. Each signal has an attribute named `Output Scope`. This scope attribute indicates whether or not the signal is to be a *local* (automatic) variable or a *global* variable. In addition, when the signal has a Global Scope, you can enter a memory address. This is just a string—31 characters maximum—that can be used to represent the

memory address, and that string is accessible through template tokens during code generation.

Note Be careful when selecting only a partial subset of block outputs as Global Scope when generating vectorized code. NI recommends that if you intend for some of the block's output to be global, you make them all global so as to preserve loops within the generated code.

Generating Code for Monitored Signals

The only difference between the code when using Global Scope signals will be the declarations of the variables used for the local block outputs. AutoCode provides the following set of TPL tokens to allow you to customize the declaration of these variables. Since monitored signals are not intended to be shared between subsystems, the global variables used to represent monitored signals are grouped by subsystem. For more information about these tokens, refer to the *Template Programming Language User Guide*.

The default template uses the following tokens to declare the variables. The visibility of the global variable declarations must include the subsystem for which they are intended.

`gtype_nmembers_i`	`gtype_b`
`gtype_members_ls`	`gtype_members_size_li`
`gtype_members_typ_pfix_ls`	`gtype_members_typ_li`
`gtype_members_memaddr_ls`	`gtype_blk_list_li`
`gtype_blk_channel_list_li`	

Note You must write template code to customize the declarations of these variables. That includes usage of the Memory Address string, as this is target/compiler specific information.

Limitations

This section identifies some of the limitations of scoped output.

Unsupported Blocks

The following list presents the blocks that do not support scoped outputs.

- Blocks that are not supported by AutoCode
- Write-to Variable Block
- DataStore
- Any block that can never have outputs, such as Text Block and Sequencer

Connection to External Output

It is possible to connect a Global Scope output to a subsystem/system external output pin. A Global Scope output cannot be used to implicitly communicate between subsystems and/or the system output. If you connect your model this way, there will be a copy from the global variable into the external output structure for that system/subsystem. That copy occurs at the end of the subsystem execution order.

Variable Block Aliasing

You cannot name a Global Scope label or name that matches the name of the variable block or %var variable. AutoCode will mangle the name of the Global Scope signal in that situation.

Monitored Signals within a Procedure SuperBlock

The implementation of monitoring signals uses global variables. That means there is only one set of global variables for each use of the procedure. By definition, that means that procedures that use monitored signals are no longer re-entrant.

Monitoring Procedure External Outputs

The external outputs of a Procedure SuperBlock cannot be safely monitored. The reason is that the concept of monitoring is in conflict with the parameterless procedure feature. Therefore, external outputs of a procedure cannot use a safe global variable and must be considered a part of a parameterless interface to that procedure.

Parameterless Procedure

A parameterless (argument-less) procedure is a procedure that uses global variable(s) to pass input(s) into and/or output(s) out of the procedure. Each input and output can be individually selected to be global or not.

Note A parameterless procedure is purely a performance optimization that requires you to make design changes in your model to get the correct code. Carefully consider using this type of procedure, as it requires significant effort to design your model correctly without the safety net of AutoCode managing the generated variables. National Instruments recommends that only advanced users of SystemBuild and AutoCode use this feature.

Specifying Parameterless Procedure Interface

Each of the inputs and outputs of a procedure can be specified with a scope. This scope indicates that the signal is to use a global variable or not. Unfortunately, it is not enough just to specify the interface. The source of the procedure's inputs—that is, the blocks whose output(s) connect to the procedure's inputs—must know the exact name(s) of the global variables so as to pass the data correctly. The goal is to eliminate any extra copies of the data when a procedure is called, and that requires the use of unsafe global variables.

An unsafe global variable is a global variable that might have multiple-writers and might require an implicit execution order to provide the correct value for a particular usage.

Caution The Analyzer and AutoCode are unable to detect sequencing flaws in a model that uses parameterless procedure signals. The result might be non-deterministic behavior of your model.

Input Specification

The specification of procedure inputs is done within the Procedure SuperBlock definition. Like basic block outputs, each input channel has an attribute called `Input Scope`. A Local Scope for an input indicates that channel will be passed to the procedure within the procedure's input (U) structure or passed on the stack (`-nouy`). A Global Scope indicates that the signal is intended to be a global variable. As such, the exact name of the global variable must be specified as the procedure's Input Name for that signal. You cannot use a Global Scope Input signal without specifying an input name for that signal.

Note NI suggests that you adopt a naming convention for all of your global variables used for parameterless procedure signals. For example, specify all of the names with a leading "g," like `gThrottleValue`.

Output Specification

Procedure outputs are specified from the basic blocks that connect to the external signal pins. Therefore, you must specify the global output of a procedure at the basic block that is the source of the output signal. After the basic block and channel is identified, select the channel's Output Scope to be Global and specify the exact name of the global variable to use within the Output Label or Name field for that signal. Notice that the normal Name/Label precedence for basic block outputs still applies. Again, a naming convention for the global variables is recommended.

Using a Parameterless Procedure

After the procedure has been defined with parameterless signals, you now have to connect signals to and from the procedure reference. The following sections describe the connections of a parameterless signal.

Global-to-Global Input Connection

The global-to-global input connection is the most optimal usage of the parameterless procedure interface. This form indicates that the source of the input directly uses the global variable used for the procedure's parameterless signal. Therefore, you must match the global variable(s) properly to connect to a parameterless procedure. The Analyzer will verify that the global variable names match.

Note You must indicate within the source block the signal connected to the procedure to be Global Scope and specify the same name for its global variable as the global variable specified for the procedure's Input Name.

Global Output Connection

You can use a global output of the procedure like any basic block with a Global Scope Output, but you cannot change the Scope of the signal at the procedure reference.

Condition Block Code Generation

The Condition Block supports the use of parameterless procedures. In the nodefault and sequential modes, the Condition Block will not buffer the global outputs of the procedure into the `R_P` (real parameter) array. For information about Condition Blocks, refer to the *Condition Block* section of Chapter 5, *Generated Code Architecture*.

Reusing a Parameterless Procedure

A parameterless procedure is a procedure that uses a set of global variables as its interface rather then a formal argument list. Therefore, you can reuse—that is, use the procedure in more than one place—but you must make sure that source blocks for the Global Scope inputs use the correct global variable. When you reuse a parameterless procedure, there now are multiple writers to the same global variable and, therefore, the sequencing of the blocks can unexpectedly affect the value of the global variable.

Note If you intend to reuse a parameterless procedure, make sure that the sequencing of the blocks that write into the global variables is correct. Also, keep in mind the same problem with using the global outputs of the procedure as well.

Generating Code for Parameterless Procedures

The only difference between the code of a parameterless procedure and one that does not use this feature is that not all of the inputs and outputs of the procedure will be formal arguments of the procedure. Global variables are generated when referring to a procedure's inputs and/or outputs. As global variables used for a parameterless procedure require visibility over all subsystems and procedures, these variables are associated with the whole model. The set of TPL tokens to access this information is therefore within the SYSTEM scope. Refer to the *Template Programming Language User Guide* for more information about these tokens.

The default template uses the following tokens to declare the variables.

`sgtype_nmembers_i`	`sgtype_b`
`sgtype_members_ls`	`sgtype_members_size_li`
`sgtype_members_typ_pfix_ls`	`sgtype_members_typ_li`
`sgtype_members_memaddr_ls`	`sgtype_blk_channel_list_li`
`sgtype_blk_list_li`	`sgtype_blk_list_idx_li`

Note You must write template code to customize the declarations of these variables, which includes usage of the Memory Address string because this is target/compiler specific information.

Issues and Limitations

This section identifies some other items that you should be aware of if you intend to use parameterless procedures.

Note Parameterless procedures require the use of global variables. All of the deterministic safety measures normally used by AutoCode are disabled for these signals. As a result, you can easily create models that exhibit non-deterministic behavior. Issues such as block sequencing and selection of variables are now your responsibility.

Communication Between Subsystems

You cannot use a Global Scope signal to bypass the subsystem interface. A copy will be required to pass the data between subsystems. If you want to pass data between subsystems using global variables, use a variable block.

Variable Blocks Versus Global Scope

A Variable Block cannot be used as a Global Scope signal into a parameterless procedure. The reason for this is that you would be implicitly writing to that variable block when the procedure is reused. You should contain the Variable Block within the procedure rather than passing it into the procedure. In addition, the SystemBuild Simulator would be unable to simulate the aliasing effect of the Variable Block and the Global Scope signal.

SystemBuild Simulator

The SystemBuild Simulator ignores the Output and Input Scope attributes. Therefore, the subtle effects of the global variables are not simulated and can mask those effects, thus misleading you to believe your model is correct while global variables in the generated code are being overwritten and corrupting the results of your model. If you correctly use the global variables by proper sequencing and selection, the Simulation should match the generated code. However, it could also be a coincidence, and there is no direct way to tell the difference.

Connection to External Output

If a signal that is used as input to a Global Scope procedure input or a Global Scope procedure output is connected to a external output pin, be aware that the external output pins will be updated at the end of that subsystem during the subsystem copy-back phase. Thus, if you reuse the Global variable representing that signal, by reusing the procedure, you will see the last value of that global signal for each of its external output connections. In this case, NI recommends copying the global variable after each use through a gain block and connect that block output to the external output pin.

Recommendations

NI recommends applying the following steps to your design if you intend to use parameterless procedures. Notice that many of these items have direct parallels to what you would expect if you were hand-coding using global variables.

Naming Convention

Adopt a model-wide naming convention for the global variables used for passing data into and out of the parameterless procedure. You might want one style for inputs and another for outputs.

Model Documentation

Use the Text Block extensively to document both the procedure definition and each procedure reference. This will help indicate what the expectation is for the procedure. Concentrate on describing the interface of the procedure as well as sequencing assumptions.

Explicit Sequencing

As an added safety measure, you might want to use the Sequencer Block to insure the proper ordering of blocks that affect global variables. This is critical when you are reusing a parameterless procedure or even the global variable itself.

Analyzer and AutoCode Warnings

The Analyzer reports questionable connectivities or situations regarding usage of the Global Scope signals. Also, AutoCode might report additional warnings during code generation. Evaluate these warnings and verify that the generated code is what you expect. You might need to change your model to fix the situation.

Changing Scope Class

To properly match Global Scope signals for parameterless procedures, you might need to use an explicit copy to resolve the naming. This is accomplished by using a unit-gain block—that is, a gain block with a gain factor of 1. Using the gain block in this way will let you match any connectivity that you require. You might want to use a custom icon for this usage of the gain block to provide a better graphical representation of the copy instead of a gain.

Command Options

The following command options force the scope of block outputs:

- `-nogscope`—This option forces all Output Scopes to be Local for all block outputs. Connection to a Input Scope that is Global results in a copy into the global variable before the procedure call.
- `-allgscope`—This option forces all Output Scopes to be Global for all block outputs while Input Scopes are forced to Local, thus disabling parameterless procedures.

A

Technical Support and Professional Services

Visit the following sections of the National Instruments Web site at ni.com for technical support and professional services:

- **Support**—Online technical support resources at ni.com/support include the following:
  - **Self-Help Resources**—For answers and solutions, visit the award-winning National Instruments Web site for software drivers and updates, a searchable KnowledgeBase, product manuals, step-by-step troubleshooting wizards, thousands of example programs, tutorials, application notes, instrument drivers, and so on.
  - **Free Technical Support**—All registered users receive free Basic Service, which includes access to hundreds of Application Engineers worldwide in the NI Discussion Forums at ni.com/forums. National Instruments Application Engineers make sure every question receives an answer.

    For information about other technical support options in your area, visit ni.com/services or contact your local office at ni.com/contact.
- **Training and Certification**—Visit ni.com/training for self-paced training, eLearning virtual classrooms, interactive CDs, and Certification program information. You also can register for instructor-led, hands-on courses at locations around the world.
- **System Integration**—If you have time constraints, limited in-house technical resources, or other project challenges, National Instruments Alliance Partner members can help. To learn more, call your local NI office or visit ni.com/alliance.

If you searched ni.com and could not find the answers you need, contact your local office or NI corporate headquarters. Phone numbers for our worldwide offices are listed at the front of this manual. You also can visit the Worldwide Offices section of ni.com/niglobal to access the branch office Web sites, which provide up-to-date contact information, support phone numbers, email addresses, and current events.

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Y